4

WINDOWS

Includes Demonstration Program Windows1

Introduction

A **window** is a user interface element. More specifically, it is an area on the screen in which the user can enter or view information. A Macintosh application uses windows for most communication with the user, from discrete interactions such as presenting and acknowledging alert boxes to open-ended interactions such as creating and editing documents. Users generally enter data in windows and your application typically lets the user save this data to a file.

The Window Manager, which, amongst other things, provides functions for managing windows, itself depends on QuickDraw. QuickDraw supports drawing into **colour graphics ports**, which are individual and complete drawing environments with independent coordinate systems. Each window represents a colour graphics port.

Your application typically creates **document windows**, which allow the user to enter and display text, graphics, or other information. 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.

Window Basics

Standard Window Elements

The Window Manager defines and supports a set of standard window elements through which the user can manipulate windows:

- ***Title Bar.*** 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. You usually display a newly created window with the title "untitled". When the user opens a saved document, you assign the document's filename to the window in which it is displayed.

- ***Close Box.*** Offers the user a quick way to close a window. The close box is sometimes called the **go-away box**.

- ***Full, Vertical, and Horizontal Zoom Boxes.*** Offer the user a quick way to choose between two different window sizes, one established by the user and one by the application.
- ***Collapse Box.*** Lets the user collapse and uncollapse a window.
- ***Size Box.*** Lets the user change the size of a window.
- ***Draggable Area.*** That part of the window's frame less the title bar.

> ***Historical Note***
>
> The vertical zoom box, the horizontal zoom box, the collapse box, and the draggable area were introduced with Mac OS 8 and the Appearance Manager.

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

Active and Inactive Windows

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

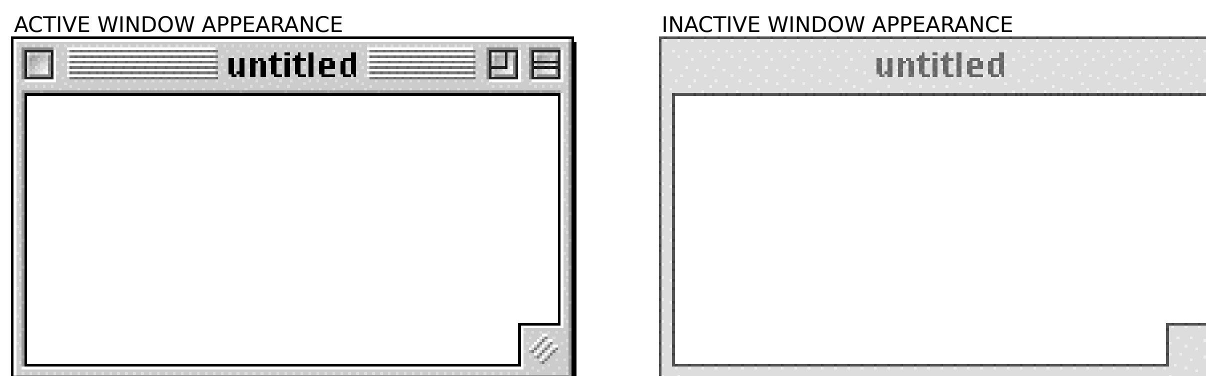


FIG 1 - APPEARANCE OF ACTIVE AND INACTIVE WINDOWS

When the user activates one of your application's windows, the Window Manager redraws the window's title bar and frame, the close box, the title text, the zoom box, the collapse box, and the size box. 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 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 as shown at Fig 1, hiding the close, zoom, collapse, and size boxes. Your application must deactivate any controls, remove highlighting from selections, and so on.

> ***Historical Note***
>
> Prior to Mac OS 8 and the Appearance Manager, your application was also required to draw (on window activation and on receipt of update events) and erase (on window deactivation) the size box via a call to DrawGrowIcon. The new Appearance-compliant window definition functions (see below) which were

introduced with Mac OS 8 and the Appearance Manager relieve your application of that responsibility and merge the size box into the window frame.

In Appearance-compliant windows, DrawGrowIcon, if called, will simply draw scroll bar delimiting lines (single lines extending left and upwards from the top and left side, respectively, of the size box), and may still be used for that purpose if required.

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.

Types of Appearance-Compliant Windows

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

- Document types.
- Dialog and alert types.
- Utility window types (sometimes referred to as 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.

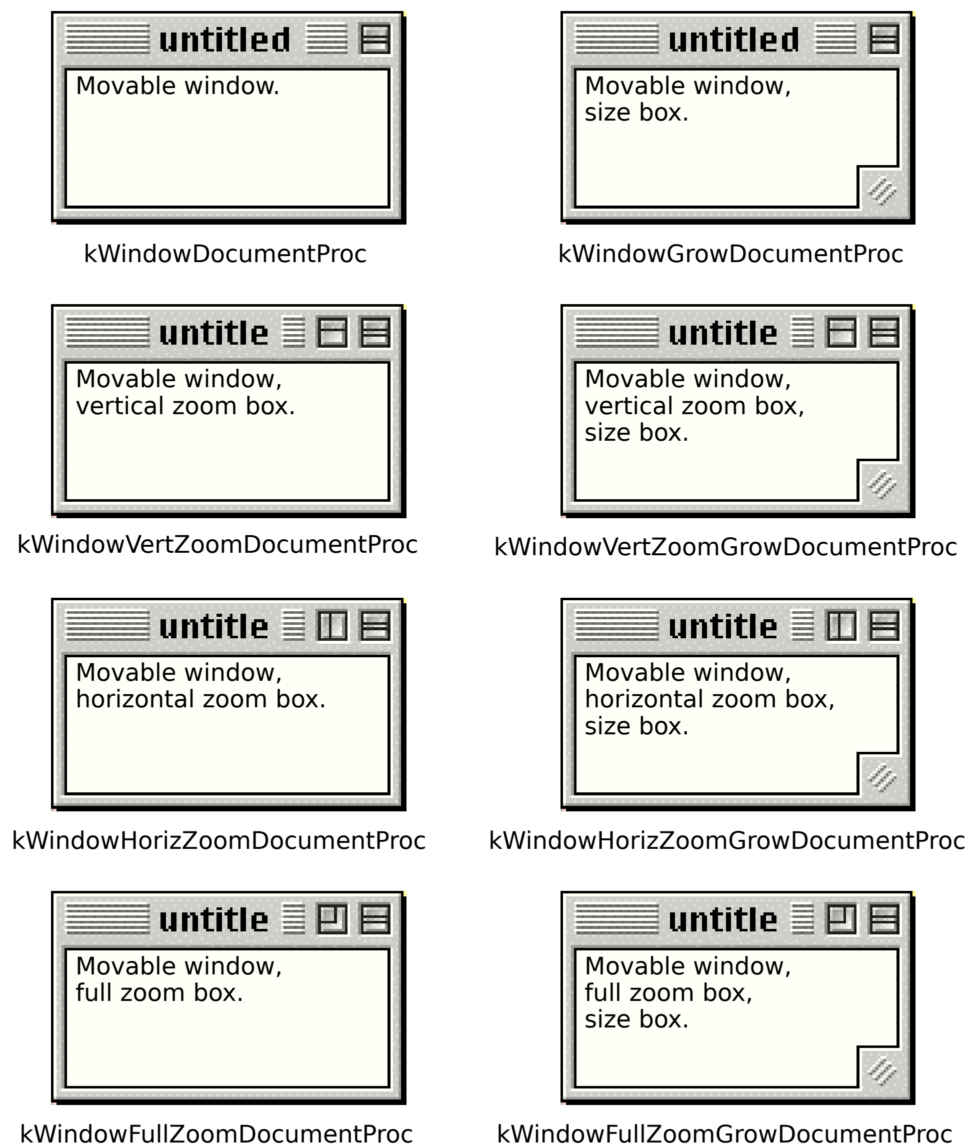


FIG 2 - WINDOW TYPES FOR DOCUMENTS

Dialog and Alert Types

Fig 3 shows the six 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 dialog boxes,)

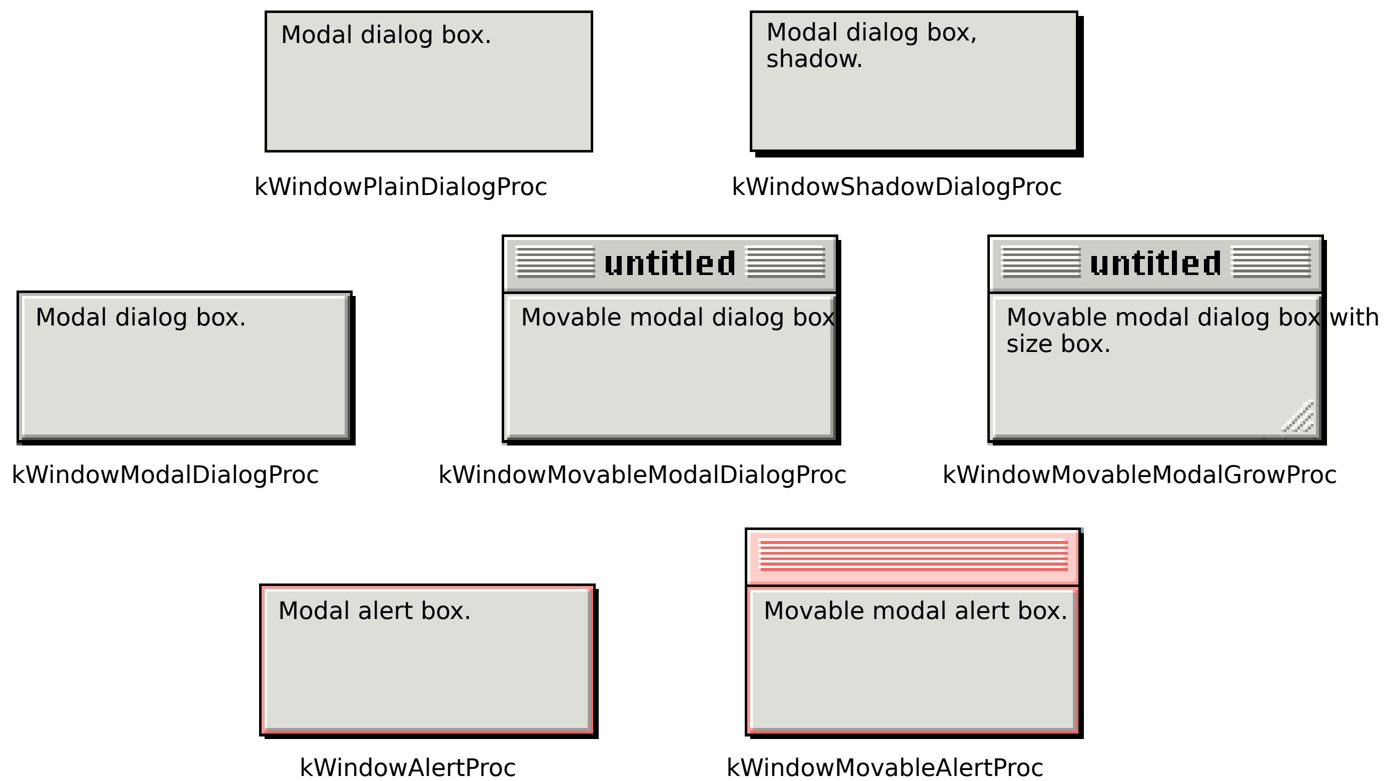


FIG 3 - WINDOW TYPES FOR DIALOGS AND ALERTS

Utility (Floating) Window Types

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

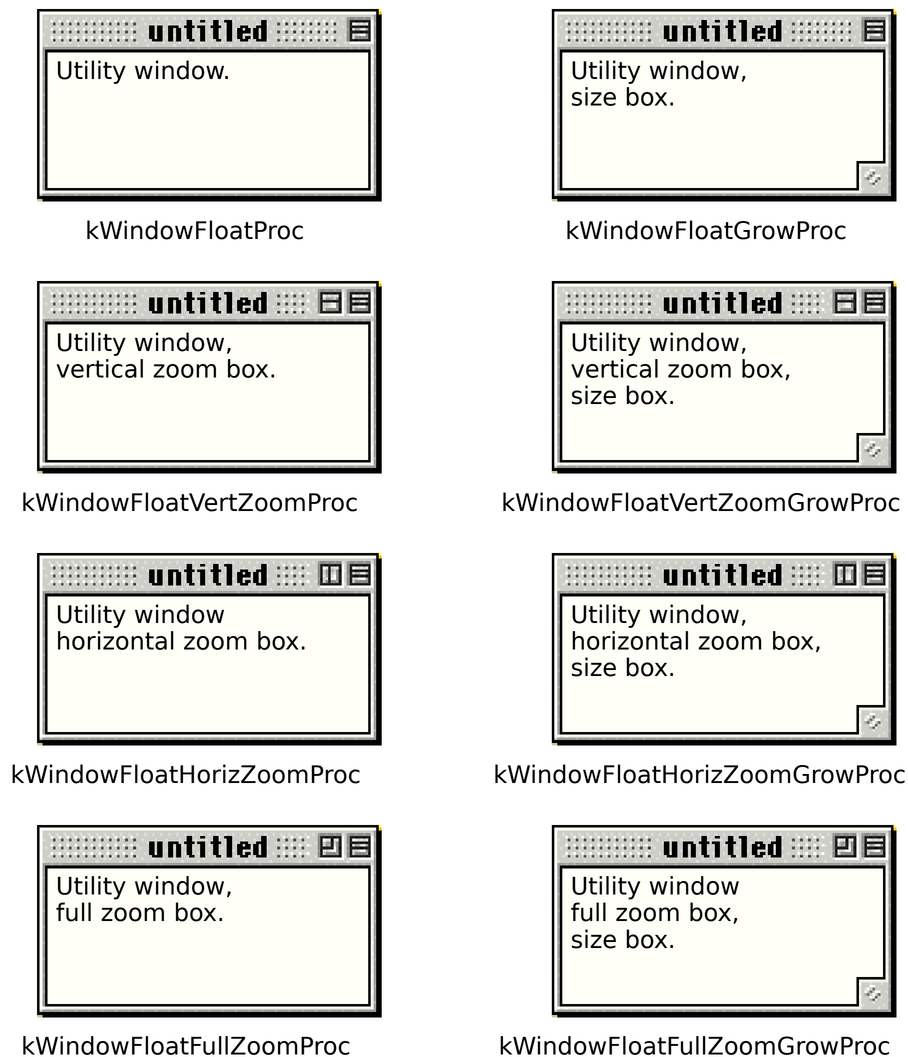


FIG 4 - WINDOW TYPES FOR UTILITY WINDOWS (TITLE B

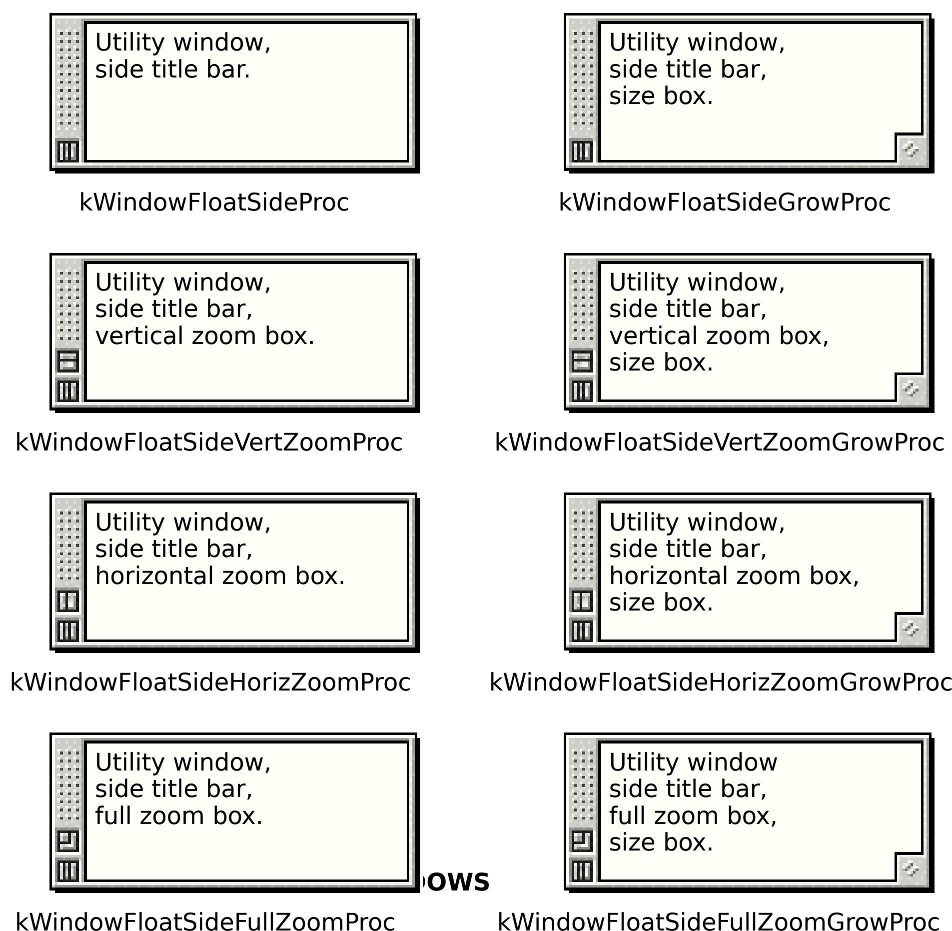


FIG 5 - WINDOW TYPES FOR UTILITY WINDOWS (PSUEDO TITL

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. The definition function draws the window's frame, draws the close, zoom, collapse, and size boxes (if any), draws the window title (if any), determines which region the cursor is in within the window, calculates the window's content and structure regions (see below), and performs any special initialisation or disposal tasks.

- ***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.

> ***Historical Note***
>
> These are the resource IDs of the new Appearance-compliant window definition functions first issued with Mac OS 8 and the Appearance Manager. The old WDEFs have resource IDs of 0 (document windows), 1 (rDocProc document window), and 124 (utility windows), and remain in the System file. The new definition functions are located in the Appearance extension in Mac OS 8.0 and 8.1, and in the System file in Mac OS 8.5.

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

Historical Note

The old pre-Mac OS 8, pre-Appearance Manager window types, and their Appearance-compliant (Fig 3) equivalents, are as follows.

Pre-Appearance	Appearance-Compliant	Description
noGrowDocProc	kWindowDocumentProc	Movable window.
documentProc	kWindowGrowDocumentProc	Movable window, size box.
zoomNoGrow	kWindowFullZoomDocumentProc	Movable window, full zoom box.
zoomDocProc	kWindowFullZoomGrowDocumentProc	Movable window, full zoom box, size box.
rDocProc	(None)	Round-cornered window.
dBoxProc	kWindowModalDialogProc	Modal dialog box.
(None)	kWindowAlertProc	Modal alert box.
movableDBoxProc	kWindowMovableModalDialogProc	Movable modal dialog box.
(None)	kWindowMovableAlertProc	Movable modal alert box.
plainDBox	kWindowPlainDialogProc	Modeless dialog box.
altDBoxProc	kWindowShadowDialogProc	Modeless dialog box, shadow.
floatProc	kWindowFloatProc	Utility.
floatGrowProc	kWindowFloatGrowProc	Utility, size box.
floatZoomProc	kWindowFloatFullZoomProc	Utility, full zoom box.
floatZoomGrowProc	kWindowFloatFullZoomGrowProc	Utility, full zoom box, size box.
floatSideProc	kWindowFloatSideProc	Utility, side title.
floatSideGrowProc	kWindowFloatSideGrowProc	Utility, side title, size box.
floatSideZoomProc	kWindowFloatSideFullZoomProc	Utility, side title, full zoom box.

floatSideZoomGrowProc	kWindowFloatSideFullZoomGrowProc	Utility , side title, size box, full zoom box.

Window Type Usage

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

Window Types For Modal Alert Boxes and Modal Dialog Boxes. Modal alert boxes and modal dialog boxes are merely special-purpose windows that require no window manipulation elements. Modal alert boxes generally use the window type kWindowAlertProc and modal dialog boxes generally use window type kWindowModalDialogProc; however, kWindowPlainDialogProc and kWindowShadowDialogProc may also be used.

Window Types For Movable Modal Alert Boxes and Movable Modal Dialog Boxes. Movable modal alert boxes and movable modal dialog boxes 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 alert boxes use the window type kWindowMovableAlertProc and movable modal dialog boxes use the window type kWindowMovableModalDialogProc.

Historical Note

Movable Modal Alert boxes were introduced with Mac OS 8 and the Appearance Manager.

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

The creation and handling of alert and dialog boxes is addressed in detail at Chapter 8—Dialogs and Alerts.

Window Regions

The Window Manager recognises the special-purpose **regions**[1] shown at Fig 6.

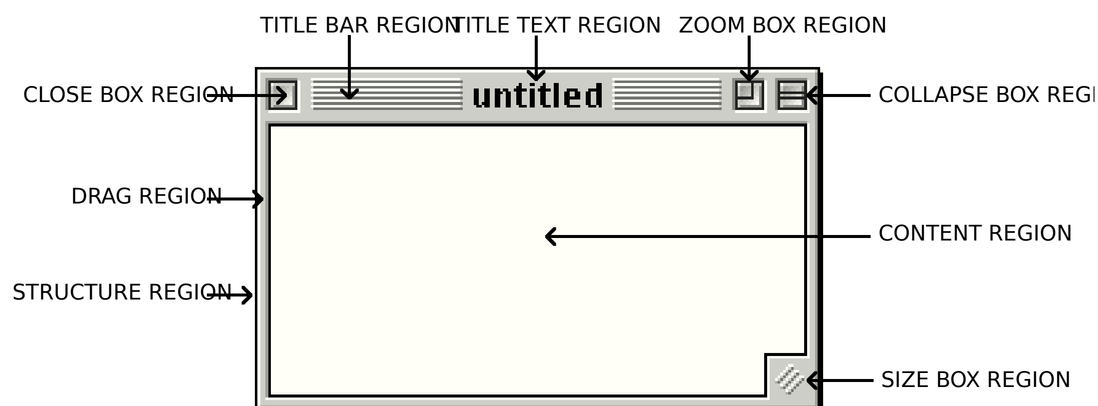


FIG 6 - WINDOW REGIONS

[1] 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 — Drawing With QuickDraw.

Handles to these 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.
Close box region	kWindowCloseBoxRgn	The area occupied by a window's close box.
Zoom box region	kWindowZoomBoxRgn	The area occupied by a window's zoom box.
Drag region	kWindowDragRgn	The draggable area of the window frame, including the title bar and window outline, but excluding the close box, zoom box, and collapse box.
Size box region	kWindowGrowRgn	The area occupied by a window's size box.
Collapse box region	kWindowCollapseBoxRgn	The area occupied by a window's collapse box.
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, the size box, and the window's controls (including scroll bars) are displayed.

Other Regions

Two other regions of relevance to the Window Manager are:

- ***The Update Region.*** The update region is a dynamic region which accumulates all areas of a window's content region which need updating (that is, re-drawing).
- ***The Gray Region.*** The entire area of the desktop, that is, the screen area that is not occupied by the menu bar, is known as the **gray region**. The Window Manager maintains a pointer to the gray region in a low-memory global variable named GrayRgn. You can retrieve a handle to the gray region with the function LMGetGrayRgn.

Controls and Control Lists

Windows may contain **controls**. The most common control in a window is the **scroll bar** (see Fig 7), 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 included in a window "belong" to that individual window and are displayed within the colour graphics port which represents that window. For each window your application creates, the Window Manager creates a **control list**, a series of entries pointing to the descriptions of controls associated with a window.

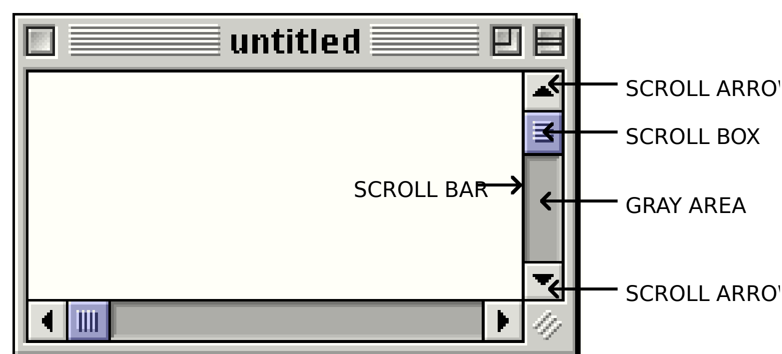


FIG 7 - SCROLL BARS

The Window List

Multiple windows from different applications may appear simultaneously on the desktop. The Window Manager tracks all windows using its own private data structure called the **window list**. Entries in the window list appear in their order on the desktop, beginning with the frontmost (active) window. When the user changes the ordering of the windows on the desktop, the Window Manager generates events telling your application to activate, deactivate and update its windows as necessary.

The Colour Graphics Port and the Colour Window Structure

The Colour Graphics Port

Each window represents a QuickDraw **colour graphics port**, which is a drawing environment with its own coordinate system. The Window Manager creates a colour graphics port when it creates the window.

The location of a window on the screen is defined in **global coordinates**, that is, coordinates which reflect the entire potential drawing space. QuickDraw recognises a coordinate plane whose origin is the upper left corner of the main screen, whose positive x-axis extends rightward and whose positive y-axis extends downward (see Fig 8). In QuickDraw functions, the horizontal offset is ordinarily labelled `h`, and the vertical offset `v`. The coordinate plane is bounded by the limits of QuickDraw coordinates, which range from -32768 to 32,767.

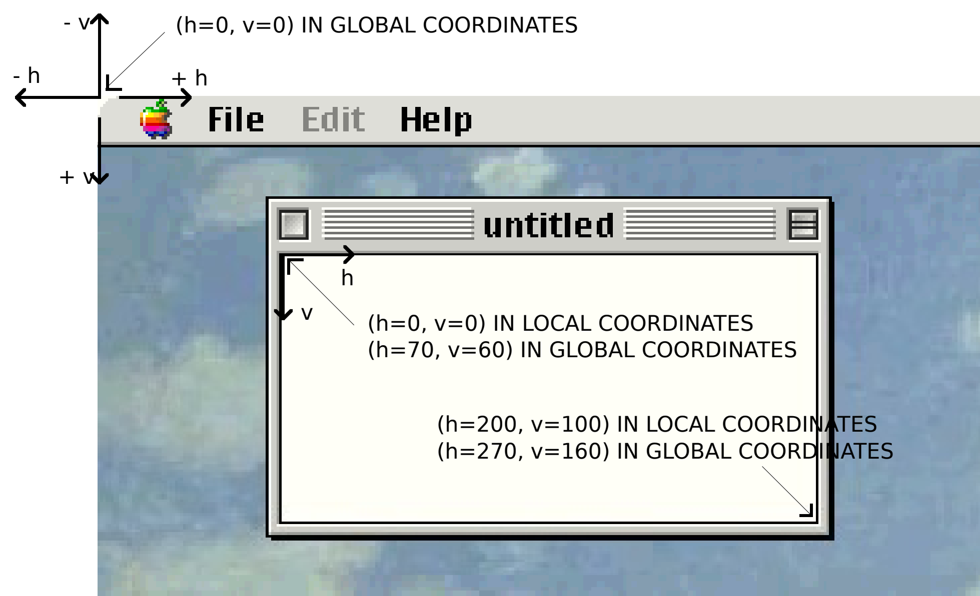


FIG 8 - A WINDOW'S LOCAL AND GLOBAL COORDINATE S

When QuickDraw creates a new colour graphics port (usually, when you create a new window), it defines a **bounding rectangle** for the port in global coordinates. Ordinarily, the bounding rectangle represents the entire area of the screen on which the window appears. The bounding rectangle is stored in the colour graphics port data structure, in the bounds field of a structure called a **pixel map** in Color QuickDraw.

The colour graphics port data structure also includes a field called portRect, which defines the rectangle to be used for drawing. In a colour graphics port representing a window, the portRect rectangle represents the window's content region. Within the port rectangle, the drawing area is described in **local coordinates**. Fig 8 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.[2]

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.

Colour Window Structure

The Window Manager stores information about a window in a **colour window structure**. A colour window structure is defined by the data type CWindowRecord:

```
struct CWindowRecord
{
    CGrafPort       port;            // Window's colour graphics port.
    short           windowKind;      // Class of window.
    Boolean         visible;         // true if window is visible.
    Boolean         hilited;         // true if window is highlighted.
    Boolean         goAwayFlag;      // true if window has close box.
    Boolean         spareFlag;       // true if window has zoom box.
    RgnHandle       strucRgn;        // Handle to structure region.
    RgnHandle       contRgn;         // Handle to content region.
    RgnHandle       updateRgn;       // Handle to update region.
    Handle          windowDefProc;   // Handle to window definition function.
    Handle          dataHandle;      // Handle to window state data structure.
    StringHandle    titleHandle;     // Handle to window's title.
    short           titleWidth;      // Title width in pixels.
    ControlRef      controlList;     // Handle to window's control list.
    CWindowPeek     nextWindow;      // Pointer to next window structure in window list.
    PicHandle       windowPic;       // Handle to an optional picture.
    long            refCon;          // Reference constant.
};
```

[2] The colour graphics port is addressed in detail at Chapter 11 — QuickDraw Preliminaries.

```
typedef struct CWindowRecord CWindowRecord;
typedef CWindowRecord *CWindowPeek;
```

> ***Historical Note***
>
> There is another window structure called the **window structure**, which is defined by the data type WindowRecord. The window structure traces its origins back to the era of black-and-white Macintoshes, and had to be used on any Macintosh when Color QuickDraw was not present. Since Color QuickDraw is always available with Mac OS 8 and the Appearance Manager, the window structure is now redundant. The only difference between a window structure and a colour window structure is that the port field is a graphics port (GrafPort) rather than a colour graphics port (CGrafPort).

It is important to note that the colour graphics port is the first field of the colour window structure and that the data type CWindowPtr is defined as a pointer to the colour graphics port, not to the colour window structure. Fields in the colour window structure are accessed using CWindowPeek, which is a pointer to a colour window structure. (CWindowPeek is rarely used, however, since you usually do not need to access or directly modify fields in a colour window structure. The Window Manager automatically updates the colour window structure when you make changes to a window, and supplies functions for changing and reading some fields of the colour window structure.)

The close box region, drag region, zoom box region, collapse box region, and size box region are not included in the colour window structure. The WDEF determines the location of those particular regions.

Compatibility

For compatibility purposes, the WindowPtr data type points to either a colour graphics port or a graphics port and the WindowPeek data type points to either a colour window structure or a window structure.

Events in Windows

As stated at Chapter 2 — Low-Level and Operating System Events, the Window Manager itself generates two types of events central to window management, namely, activate events and update events.

One of the more basic functions of the Window Manager is to report where the cursor is when the application receives a mouse-down event. As was also stated at Chapter 2, the Window Manager function FindWindow tells your application whether the cursor is in a window and, if it is in a window, in exactly which window and which part of that window. FindWindow is thus used as a first filter for mouse-down events, separating events which merely affect the window display from events which manipulate data.

Creating Your Application's Windows

You typically create document and utility windows from resources of type 'WIND', although you can create them programmatically using the function NewCWindow.

'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 utility windows.

Structure of a Compiled 'WIND' Resource

Fig 9 shows the structure of a compiled 'WIND' resource and how it "feeds" the colour window structure.

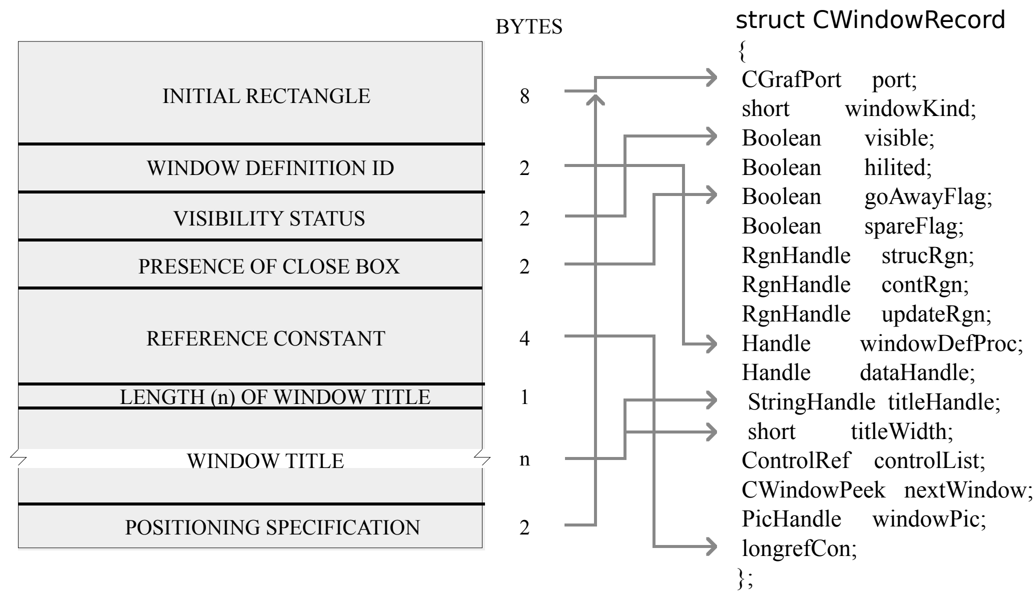


FIG 9 - STRUCTURE OF A COMPILED WINDOW ('WIND') RESOURC

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

Field	Description
INITIAL RECTANGLE	The upper-left and lower-right corners, in global coordinates, of a rectangle that defines the initial size and placement of the window's content region. Your application can change this rectangle before displaying the window, either programmatically or through an optional **positioning code** (see below).
WINDOW DEFINITION ID	The window's definition ID, which incorporates the resource ID of the WDEF that will handle the window and an optional variation code.
VISIBILITY STATUS	A specification that determines whether the window is visible or invisible. This characteristic controls only whether the Window Manager displays the window, not necessarily whether the window can be seen on the screen. (A visible window entirely covered by other windows, for example, is "visible" even though the user cannot see it.) You typically create a new window in an invisible state, build the content area of the window, and then display the completed window.
PRESENCE OF CLOSE BOX	A specification that determines whether or not the window has a close box. The Window Manager draws the close box when it draws the window frame. The window type specified in the second field determines whether a window can support a close box; this field determines whether the close box is present.
REFERENCE CONSTANT	A reference constant which your application can use for whatever data it needs to store. When it builds a new colour window structure, the Window Manager stores, in the refCon field, whatever value you specify in this field. You can also set the refCon field of the colour window structure programmatically via a call to SetWRefCon.
WINDOW TITLE	A Pascal string that specifies the window's title.
POSITIONING SPECIFICATION	An optional positioning specification that overrides the window position established by the rectangle in the first field. The positioning constants (see below) are convenient when the user is creating new documents or when you are handling your own dialog boxes and alert boxes. When you are creating a new window to display a previously saved document, however, you should display the new window in the same rectangle as that in which it was previously displayed.

Positioning Specification

The constants for the positioning specification field are as follows:

Constant	Value	Meaning
kWindowDefaultPosition	0x0000	Use initial location.
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 10 shows a 'WIND' resource being created with Resorcerer.

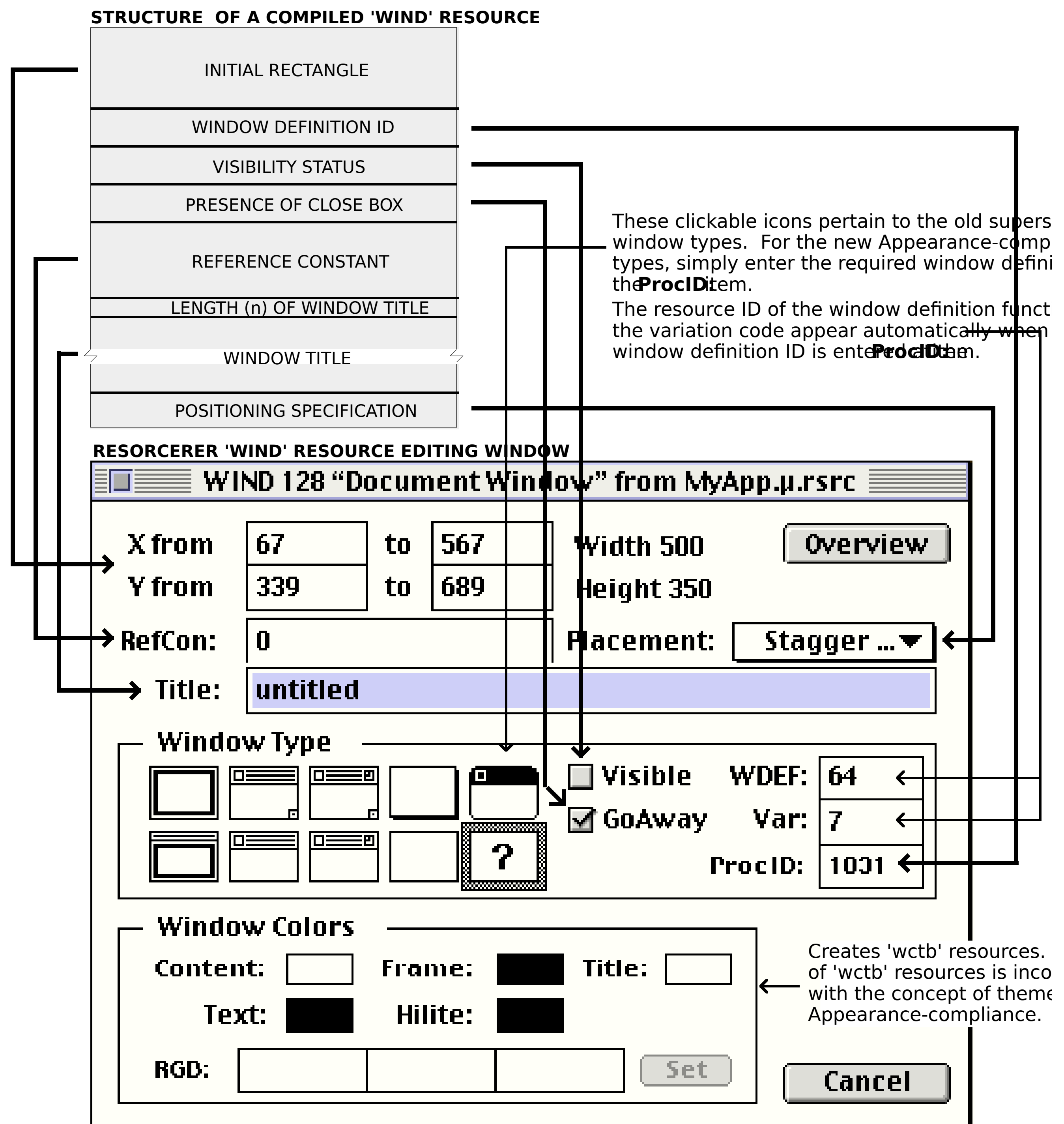


FIG 10 - CREATING A 'WIND' RESOURCE USING RESORCERER

Creating the Window From the 'WIND' Resource

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

You can allow GetNewCWindow to itself allocate memory for your colour window structure; however, memory fragmentation effects will be minimised by allocating the memory yourself from a non-relocatable block allocated for such purposes during your application's initialisation function, and then passing the pointer to GetNewCWindow.

Historical Note

The 'wctb' Resource. Prior to the introduction of the Appearance-compliant WDEFs, the colours of the various elements of a window were controlled by the **window colour table**, which contained a series of **part codes** for different window elements, together with the RGB (red-green-blue) values associated with each part. Applications typically used the default colour table; however, it was possible to explicitly control the colours used in a window by creating a window colour table ('wctb') resource with the same resource ID as the window's 'WIND' resource. The Appearance-compliant WDEFs ignore all information in the window colour table structure except the field that controls the background colour for the window's content region.

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 — Introduction to Controls).

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 saved document:

- If you are creating a window because the user is creating a new document, you can display the window immediately by calling ShowWindow. (This change in visibility adds to the update region and triggers an update event. Your application should respond to the update event by calling its own function for drawing the content region.)

- If you are creating a new window to display a saved document, you should retrieve the user's data before displaying the window.

Positioning a New Document Window on the DeskTop

New document windows should be placed just below and to the right of the last document window in which the user was working. On Macintoshes with a single screen, positioning windows is fairly straightforward. The first new document should be positioned on the upper-left corner of the desktop and each additional new document window is opened with its upper-left corner below and to the right of the upper-left corner of its predecessor. If the user closes one or more documents, subsequently opened windows should be located in the vacated positions.

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.

Positioning a Saved Document Window on the DeskTop

When you open a saved document, you should replicate the size and location of the window as it was when the document was last saved. When the user saves a document, you must therefore 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)**.

Some explanation of user state and standard state is necessary. The user state is the last size and location the user, through sizing and dragging actions, established for a window. The standard state is the size and location that your application determines is the most convenient size for the window. For windows with full zoom boxes, this typically the gray area of the screen minus three pixels all round.

The user and standard states are stored in the **state data structure**, whose handle is assigned to the dataHandle field of the colour window structure:

```
struct WStateData
{
   Rect userState;        // Size and location established by user.
   Rect stdState;         // Size and location established by application.
};

typedef struct WStateData WStateData;
typedef WStateData *WStateDataPtr, **WStateDataHandle;
```

Returning to the matter of saving the user state and the current state of the window, for windows with full zoom boxes you typically store this data as a custom resource in the resource fork of the document file. The following is an application-defined data type which will support this process by storing the user state rectangle and current zoom state while the document remains open:

```
typedef struct
{
   Rect       userStateRect;      // User state rectangle.
   Boolean    zoomState;          // Window state: true = standard state, false = user state.
} windowState;

typedef windowState *windowStatePtr;
typedef windowStatePtr *windowStateHdl;
```

This structure can be transformed into an application-defined resource which may then be stored in the resource fork of the document when the user saves the document.[3]

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.

As stated at Chapter 2 — Low-Level and Operating System Events, if the window contains a static display such as a picture, you can let the Window Manager take care of updating the content region by assigning a handle to the picture in the windowPic field of the colour window structure.

Managing Multiple Windows

Your application is likely to have multiple windows open on the desktop at once (perhaps one or more document windows and one or more dialog boxes) and it will need to keep track of them all.

You can use different strategies for keeping track of windows, including different kinds of windows. As previously stated, the refCon field in the colour window structure is set aside specifically for use by applications and can be used to store different kinds of data, such as a number representing a window type or a handle to a structure containing data relating to window management.

As an example, the refCon field could hold a number representing the type of dialog box or, in the case of document windows, a handle to an application-defined **document structure.** The document structure might typically hold a handle to the text being edited,

3 The demonstration program MoreResources.c at Chapter 17 — More on Resources shows how to save the window state to the resource fork of a document file.

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 this example application-defined document structure:

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

typedef docStructure *docStructurePtr;
typedef docStructurePtr *docStructureHdl;
```

For dialog boxes, a value of, say, 20 in the refCon field might specify a modeless dialog box which accepts input for the Find command, while a value of, say, 21 might specify a modeless dialog box that accepts input for a spelling checker. These reference constants could then control branching to application-defined window management functions specific to the particular dialog concerned.

Handling Events

Handling Mouse Events

When your application is active, it receives notice of all mouse-down events in the menu bar or in one of its windows. When it receives a mouse-down event, your application should call FindWindow to ascertain which window the mouse-down occurred in and to map the cursor location to a window region. The application should then take the appropriate action based on which window, and in which region of that window, the mouse-down occurred.

Mouse-Downs in Inactive Windows

When you receive a mouse-down event in an inactive document window or modeless dialog box, and if the active window is a document window or a modeless dialog box, you should call SelectWindow, passing it the window pointer. 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 box, no action is required by your application. Modal and movable modal alert and dialog boxes are handled by the ModalDialog function, which does not pass the event to your application.

Handling Keyboard Events

Whenever your application is the foreground process, it receives key-down events for all keyboard activity (except, of course, for the standard and user-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

The Window Manager maintains an update region, which represents the parts of your content region that have been affected by changes to the desktop. The Event Manager continually scans the updateRgn fields of the window structures of all the windows on the desktop. 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 ensures that it does not accidentally draw into other windows by clipping all screen drawing to the **visible region** of the window's colour graphics port. The visible region is that part of a colour graphics port that is actually visible on screen, that is, the part that is not covered by other windows. The Window Manager stores a handle to the visible region in the visRgn field of the colour graphics port structure.

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 11, 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 11, EndUpdate restores the visible region of the colour graphics port to the full visible region.

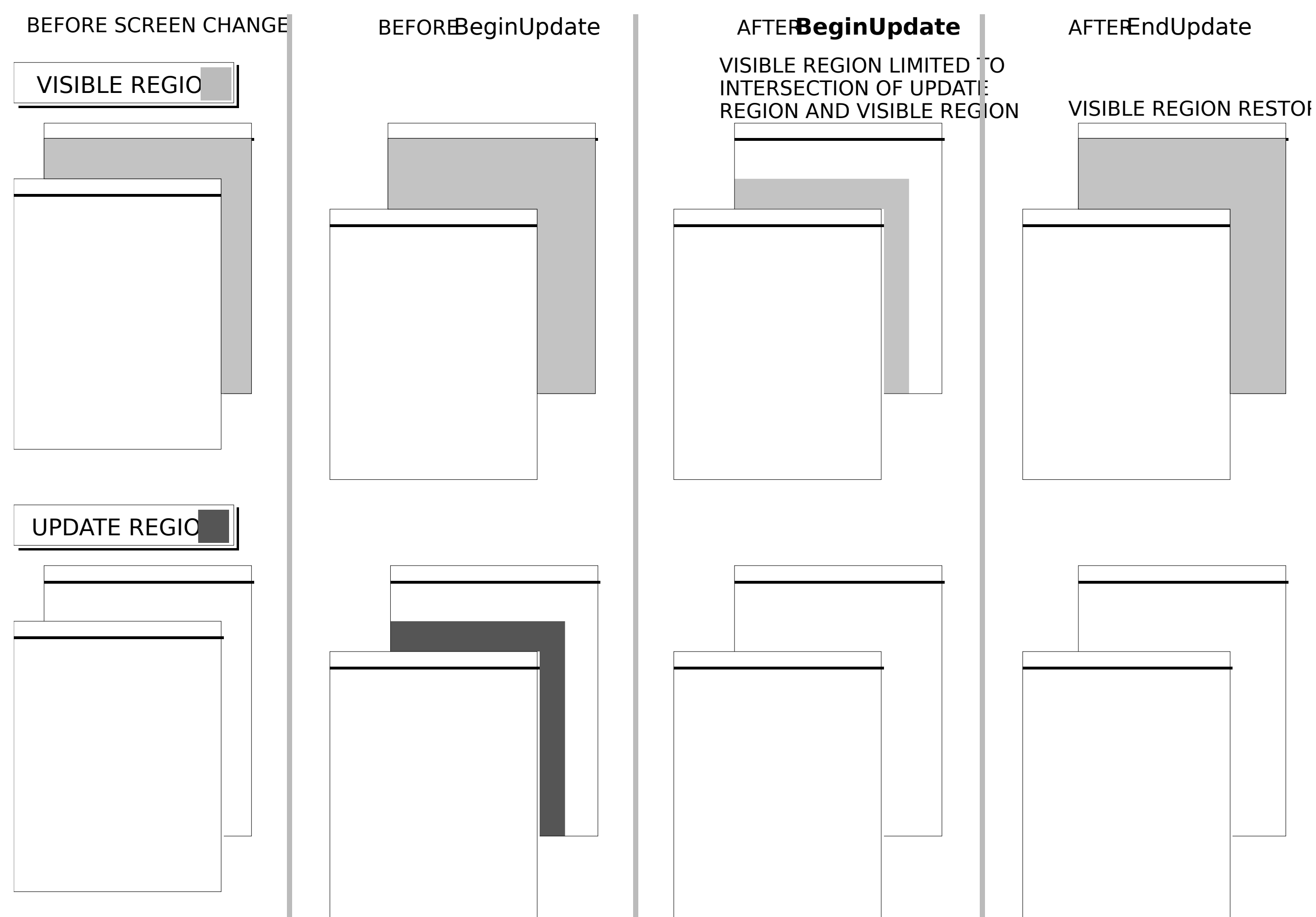


FIG 11 - EFFECTS OF BeginUpdate AND EndUpdate ON VISIBLE AND UPDATE

The reason for these update region/visible region machinations is that the handle to the update region is stored in the window structure's updateRgn field while the handle to the visible region is stored in the colour graphics port structure's visRgn field. QuickDraw

knows the colour graphics port structure intimately, but knows nothing about the window structure or its updateRgn field. 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 InValRect to add an area 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 an area from the update region by calling ValidRect 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.

Type-Dependent Update Functions

An application-defined update function should typically first determine whether the type of window being updated is a document window or some other application-defined window. If the window is a document window, an application-defined document window updating function should be called. If the window is a modeless dialog box, an application-defined 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 the direction of that change (that is, whether the window is to be activated or deactivated).

Your application typically triggers activate events itself by calling SelectWindow following a mouse-down event. SelectWindow brings the selected window to the front, removes highlighting from the previously selected window and adds highlighting to the selected window. It then generates two activate events, the first to tell your application to deactivate the previously active window and the second to activate the newly activated window.

When your application receives the event for the window about to be made inactive, it should hide the 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, adding the insertion point in its former location or highlighting previously highlighted sections as appropriate.

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

Manipulating Windows

Moving 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 draws a dotted outline of the window on the screen and moves the outline as the user moves the mouse. When the user releases the mouse, the application should call MoveWindow, which redraws the window in its new location.

Zooming a Window

Windows With Full Zoom Boxes

The zoom box allows the user to alternate quickly between two window sizes and positions. These two sizes and positions are 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 your application does not supply an initial user state, the user state is simply the size and location of the window when it was created, until the user resizes it.
- The standard state is the window size and location that your application considers most convenient. Typically, this might be the screen gray area minus three pixels all round. In a word-processing program, however, a standard state window might show a full page, if possible, or a page of full width and as much length as will fit on the screen. If the user changes the page size using the print Style dialog box, the application might adjust the standard state to reflect the new page size.
- If your application does not define a standard state, the Window Manager will automatically set it to the entire gray region of the main screen minus a three-pixel border on all sides. The user cannot change a window's standard state.
- The user and standard states are stored in a structure whose handle appears in the dataHandle field of the colour window structure. The Window Manager sets the initial values of the userState and stdState fields when it fills in the window structure and it updates the userState whenever the user resizes the window.

When the user presses the mouse button with the cursor in the zoom box, FindWindow "knows" whether the window is in the user state (zoomed-in) or the standard state (zoomed-out). When the window is in the standard state, FindWindow returns inZoomIn, meaning that the window is to be zoomed "in" to the user state. When the window is in the user state, inZoomOut is returned, meaning that the window is to be zoomed "out" to the standard state.

When FindWindow returns either inZoomIn or inZoomOut, your application should call TrackBox to handle highlighting of the zoom box and to determine whether the cursor is inside or outside the zoom box when the button is released. If TrackBox returns true, your application should call ZoomWindow to resize the window, following which it should redraw the content region.

Windows With Vertical or Horizontal Zoom Boxes

For windows with vertical or horizontal zoom boxes, you will typically want to change the size of the window when the zoom box is clicked, but not the location. This means that your application will need to define both the standard state and the user state, setting both states according to the current position of the window. You will thus need to determine the current location of the window, and set the standard and user states, immediately before the call to ZoomWindow.

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

When the user presses the mouse button in the size box, your application should call GrowWindow. This function displays a **grow image**, a dotted outline of the window frame and scroll bar area which expands and contracts as the user drags the size box.

To avoid unmanageably large or small windows, you supply upper and lower size limits when you call GrowWindow. The sizeRect parameter of GrowWindow specifies the upper and lower size limits in a single structure of type Rect. Note that the values in the structure represent window dimensions, *not* screen coordinates:

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

Most applications specify a minimum size big enough to include all parts of the structure area and the scroll bars. Because the user cannot move the cursor beyond the edges of the screen, you can safely set the maximum size to the largest possible rectangle.

When the user releases the mouse button, GrowWindow returns a long integer which describes the window's new height (in the high-order word) and width (in the low-order word). A value of zero indicates that the window size did not change. When GrowWindow returns a value other than zero, you call SizeWindow to resize the window.

When the mouse-button is released and GrowWindow returns a non-zero value, the application-defined function for re-sizing windows should call SizeWindow to draw the window in its new size. The scroll bars and window contents should then be adjusted to the new size.

Closing a Window

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

When the user clicks in the close box, 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, your application should invoke its function for closing down the window.

The specific steps you take when closing a window depend on what kind of information the window contains and whether the contents need to be saved. The application-defined function should cater for different types of windows, that is, modeless dialog boxes (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 16 — Files.)

DisposeWindow and CloseWindow

DisposeWindow removes a window from the screen, removes it from the window list, and discards all of its data storage, including the window structure. DisposeWindow should be used if you allowed the system to allocate storage for the window structure, that is, if you passed NULL as the wStorage parameter in the NewCWindow or GetNewCWindow call.

CloseWindow removes a window from the screen, removes it from the window list, and discards its data storage except for the window structure. CloseWindow should be used when you have allocated storage for the window structure manually, that is, if you created a nonrelocatable block for the window structure and passed the pointer as the wStorage parameter in the NewCWindow or GetNewCWindow call. In this case, the nonrelocatable block containing the window structure must be disposed of separately.

Hiding and Showing a Window

Whenever the user clicks the close box, you ordinarily remove the window from the screen. Sometimes, however, you might find it more convenient to merely hide the window instead of removing its data structures. If your application includes, for example, a Find modeless dialog box which searches for a string, you might want to keep its structures in memory as long as the user is working. In this case, a click on the close box should simply hide the window through a call to HideWindow. Then, when the user next chooses the Find command, the dialog box is already available, in the same location and with the same text as when it was last used.

ShowWindow will make the window visible and SelectWindow will make it the active window.

Providing Help Balloons

Help Balloons —'hrct' and 'hwin' Resources

The 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.

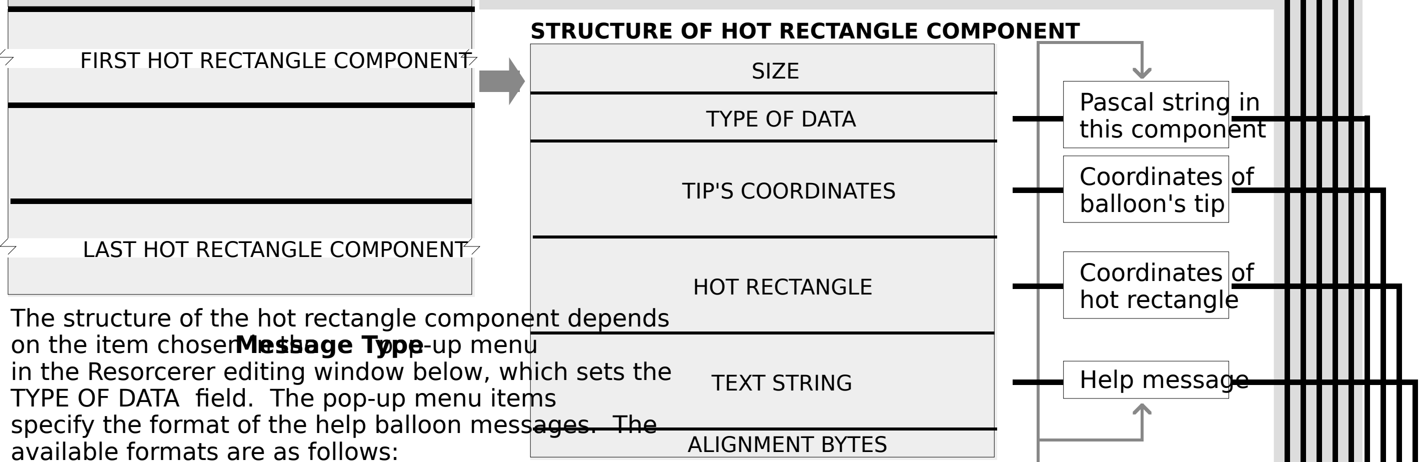
Help balloons for static document and utility windows are defined in 'hrct' and 'hwin' resources.

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 12 shows an 'hrct' resource being created using Resorcerer.

STRUCTURE OF A COMPILED hrct' RESOURCE — Header componen

HELP MANAGER VERSION	Help Manager version
OPTIONS	A number of options. 0, below, is irrelevant. 1 is not relevant for static wind 3 relate to the three different ways that the Help Manager draws and remove 4 is used in 'hwin' resources only.
BALLOON DEFINITION FUNCTION	Resource ID of the window definition function (WDEF) used for drawing help The standard WDEF's resource ID is 126. This can be specified by 0 in Resor
VARIATION CODE	Variation code for WDEF. Governs the location of the balloon's tip.
HOT RECTANGLE COMPONENT COUNT	The number of hot rectangle components defined in the rest of this resource



The structure of the hot rectangle component depends on the item chosen at 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 i... xam
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

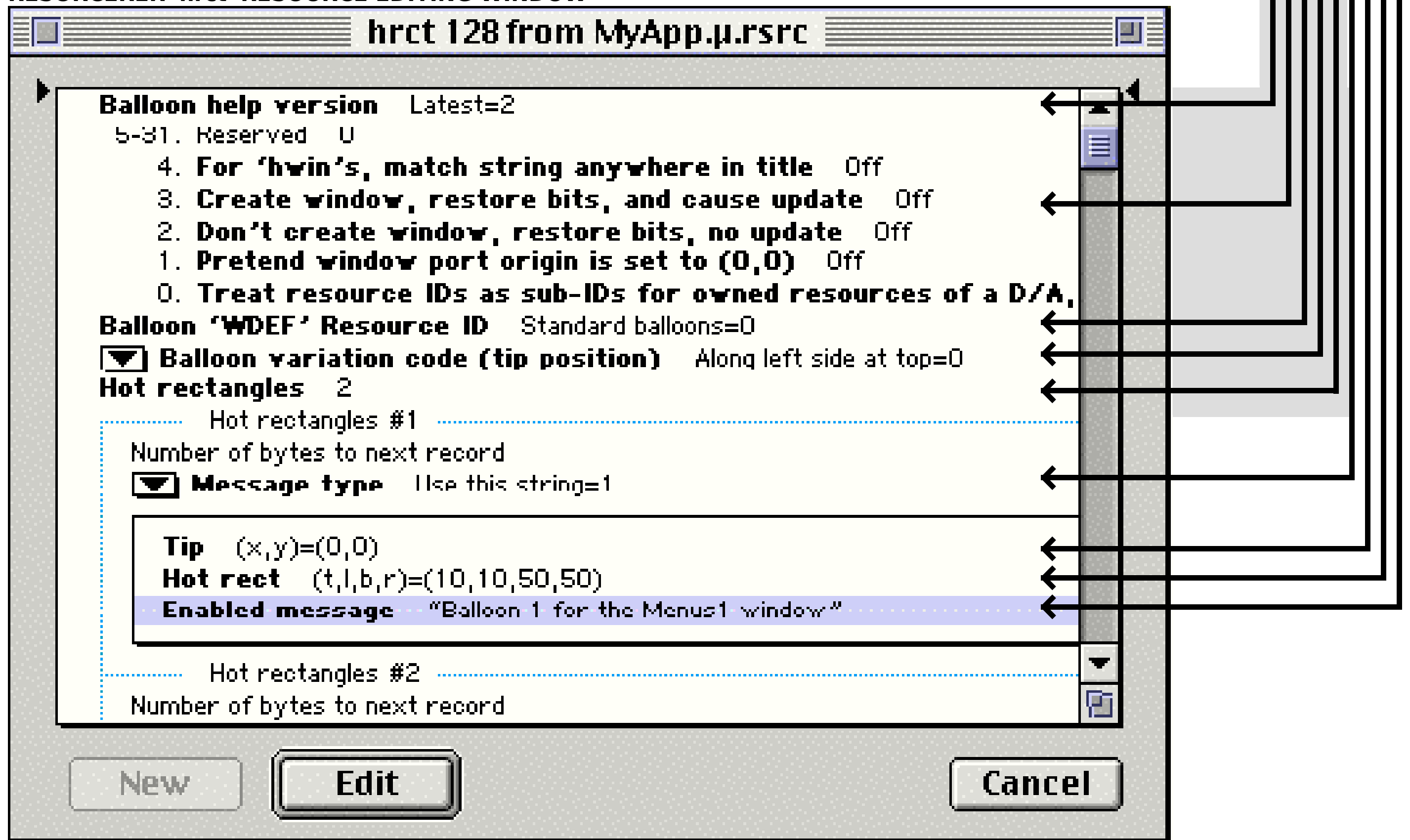
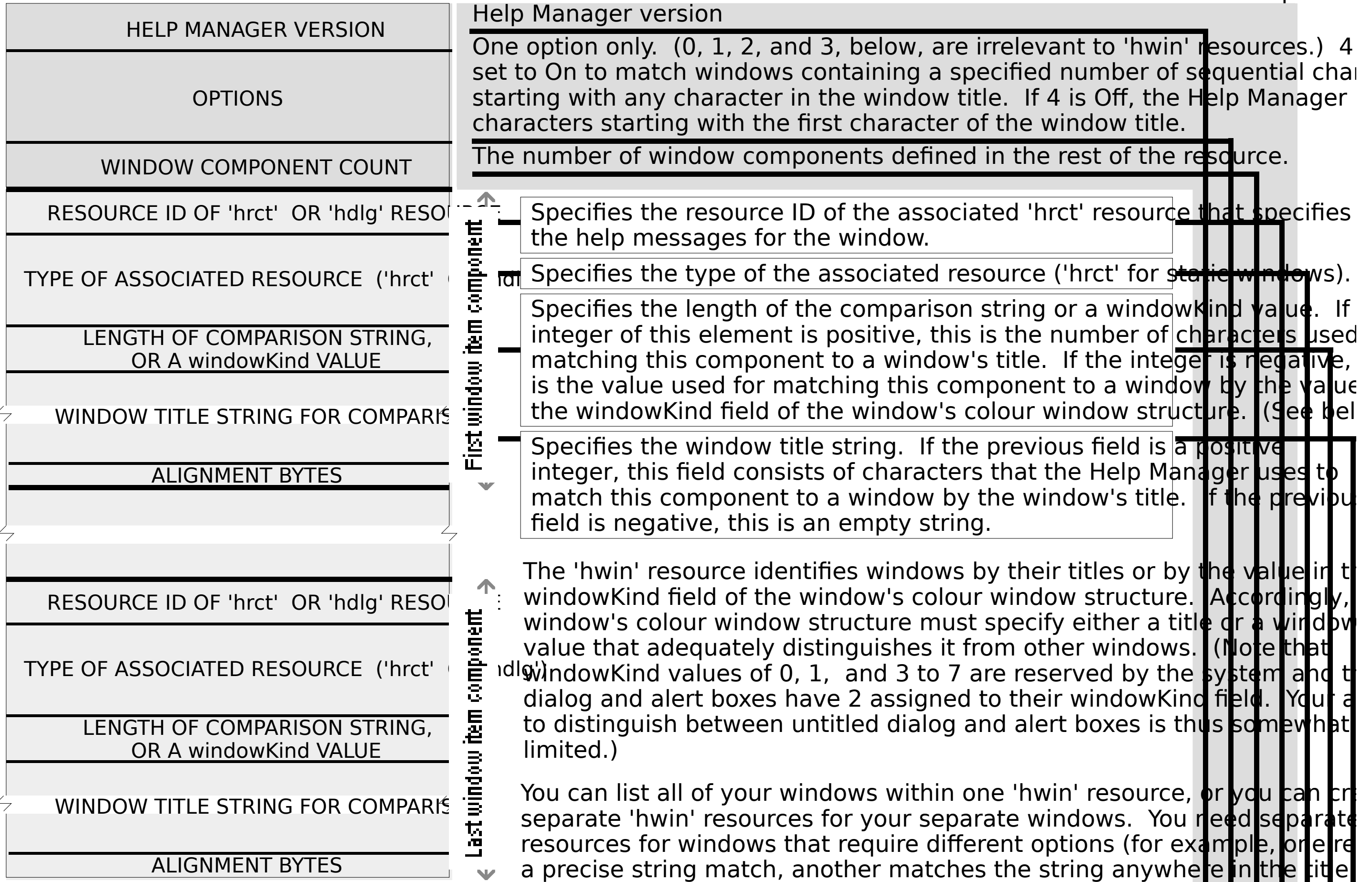


FIG 12 - CREATING AN 'hrct' RESOURCE USING RESORCERER

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 13 shows an 'hwin' resource being created using Resorcerer.



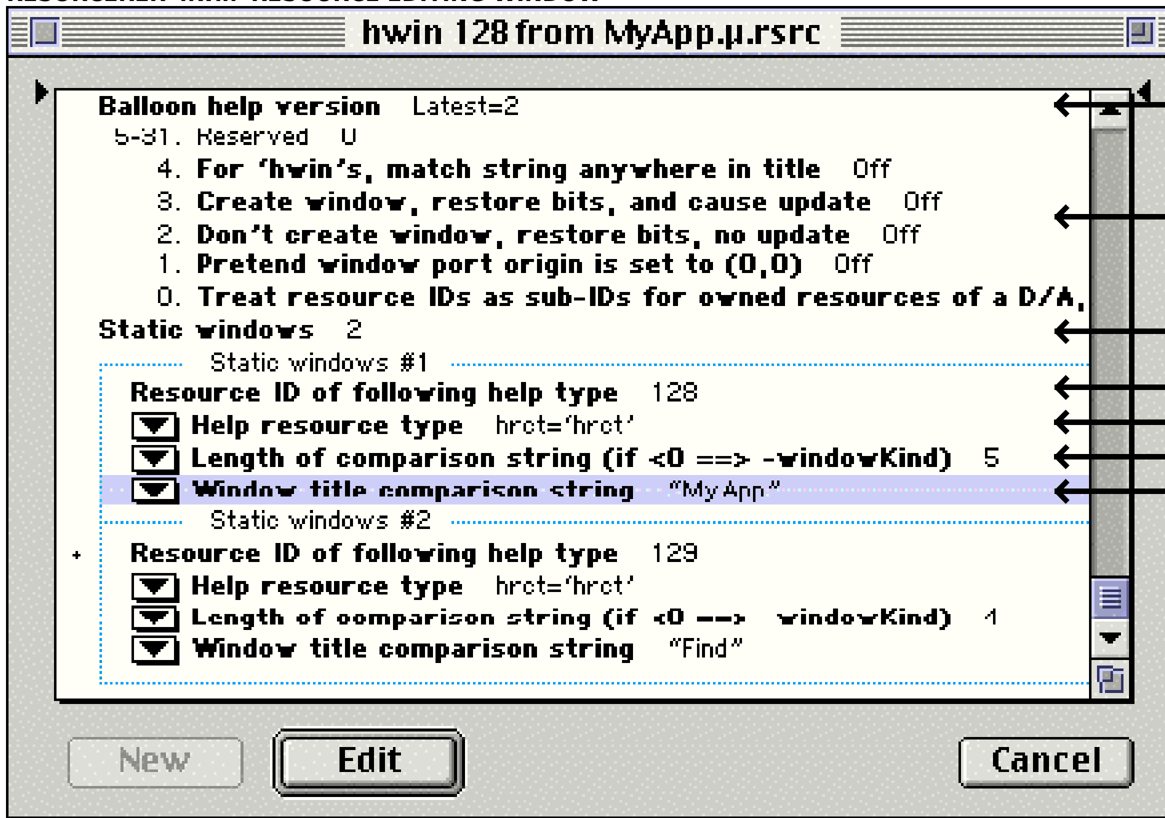


FIG 13 - CREATING AN 'hwin' RESOURCE USING RESORCERER

Main Window Manager Constants, Data Types and Functions

In the following:

- The constants, data types, and functions introduced with Mac OS 8 and the Appearance Manager are shown on a light gray background.
- Those older constants, data types and functions affected by the introduction of Mac OS 8 and the Appearance Manager, but which may still be used in certain circumstances, are shown against a black background.

Constants

Theme-Compliant 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
kWindowFloatProc                     = 1057    // Utility (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
```

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
```

Window Regions

```
kWindowTitleBarRgn      = 0
kWindowTitleTextRgn     = 1
kWindowCloseBoxRgn      = 2
```

```
kWindowZoomBoxRgn          = 3
kWindowDragRgn             = 5
kWindowGrowRgn             = 6
kWindowCollapseBoxRgn      = 7
kWindowStructureRgn        = 32
kWindowContentRgn          = 33
```

Data Types

Colour Window Structure

```
struct CWindowRecord
{
    CGrafPort      port;            // Window's colour graphics port.
    short          windowKind;      // Class of window.
    Boolean        visible;         // true if window is visible.
    Boolean        hilited;         // true if window is highlighted.
    Boolean        goAwayFlag;      // true if window has close box.
    Boolean        spareFlag;       // true if window has zoom box.
    RgnHandle      strucRgn;        // Handle to structure region.
    RgnHandle      contRgn;         // Handle to content region.
    RgnHandle      updateRgn;       // Handle to update region.
    Handle         windowDefProc;   // Handle to window definition function.
    Handle         dataHandle;      // Handle to window state data structure.
    StringHandle   titleHandle;     // Handle to window's title.
    short          titleWidth;      // Title width in pixels.
    ControlRef     controlList;     // Handle to window's control list.
    CWindowPeek    nextWindow;      // Pointer to next window structure in window list.
    PicHandle      windowPic;       // Handle to an optional picture.
    long           refCon;          // Reference constant.
};
typedef struct CWindowRecord CWindowRecord;
typedef CWindowRecord *CWindowPeek;
```

State Data Structure

```
struct WStateData
{
    Rect   userState;    // User state.
    Rect   stdState;     // Standard state.
};
typedef struct WStateData WStateData;
typedef WStateData *WStateDataPtr, **WStateDataHandle;
```

Functions

Initializing the Window Manager

```
void         InitWindows(void);
```

Creating Windows

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

Naming Windows

```
void         GetWTitle(WindowPtr theWindow,Str255 title);
void         SetWTitle(WindowPtr theWindow,ConstStr255Param title);
```

Displaying Windows

```
void         DrawGrowIcon(WindowPtr theWindow);
void         SelectWindow(WindowPtr theWindow);
void         ShowWindow(WindowPtr theWindow);
void         HideWindow(WindowPtr theWindow);
void         ShowHide(WindowPtr theWindow,Boolean showFlag);
void         HiliteWindow(WindowPtr theWindow,Boolean fHilite);
void         BringToFront(WindowPtr theWindow);
void         SendBehind(WindowPtr theWindow,WindowPtr behindWindow);
```

Retrieving Mouse Information

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

Moving Windows

```
void         MoveWindow(WindowPtr theWindow,short hGlobal,short vGlobal,Boolean front);
void         DragWindow(WindowPtr theWindow,Point startPt,const Rect *boundsRect);
long         DragGrayRgn(RgnHandle theRgn,Point startPt,const Rect *boundsRect,const Rect
             *slopRect,short axis,DragGrayRgnProcPtr actionProc);
```

Resizing Windows

```
void         SizeWindow(WindowPtr theWindow,short w,short h,Boolean fUpdate);
long         GrowWindow(WindowPtr theWindow,Point startPt,const Rect *bBox);
```

Zooming Windows

```
Boolean      TrackBox(WindowPtr theWindow,Point thePt,short partCode);
void         ZoomWindow(WindowPtr theWindow,short partCode,Boolean front);
```

Closing and Deallocating Windows

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

Maintaining the Update Region

```
void         BeginUpdate(WindowPtr theWindow);
void         EndUpdate(WindowPtr theWindow);
void         InvalRect(const Rect *badRect);
void         InvalRgn(RgnHandle badRgn);
void         ValidRect(const Rect *goodRect);
void         ValidRgn(RgnHandle goodRgn);
```

Setting and Retrieving Other Window Characteristics

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

Retrieving Window Information

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

Collapsing Windows

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

Manipulating the Desktop

```
void         SetDeskCPat(PixPatHandle deskPixPat);
void         GetWMgrPort(GrafPtr *wPort);
void         GetCWMgrPort(CGrafPtr *wMgrCPort);
RgnHandle    LMGetGrayRgn(void);
```

Demonstration Program

```
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊
// Windows1.c
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊
//
// 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, 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.
//
// •   Demonstrates the provision of balloon help for static windows.
//
// The program utilises the following resources:
//
// •   An 'MBAR' resource, and 'MENU' resources for Apple, File, Edit and Windows menus
//     (preload, non-purgeable).
//
// •   A 'WIND' resource (purgeable) (initially not visible).
//
// •   An 'ALRT' resource and 'DITL' resource for use by Stop Alerts (purgeable).
//
// •   A 'STR#' resource containing strings for the window title and error Alert box
//     (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, doesActivateOnFGSwitch,
//     and is32BitCompatible flags set.
//
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊

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

#include <Appearance.h>
#include <Devices.h>
#include <Sound.h>
#include <ToolUtils.h>

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

#define mApple              128
#define  iAbout           1
#define mFile               129
#define mEdit               130
#define  iNew               1
#define  iClose           4
#define  iQuit            11
#define mWindows            131
#define rNewWindow          128
```

```
#define rMenubar        128
#define rStringList     128
#define  sUntitled    1
#define  eMaxWindows2
#define  eFailWindow  4
#define  eFailMenus   5
#define  eFailMemory  6
#define rPixelPattern  128
#define kMaxWindows     10
#define MAXLONG              0x7FFFFFFF

#define MIN(a,b)         ((a) < (b) ? (a) : (b))

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

Boolean       gDone;
Boolean       gInBackground;
Ptr           gPreAllocatedBlockPtr;
SInt32      gUntitledWindowNumber     = 0;
SInt32      gCurrentNumberOfWindows = 0;
WindowPtr   gWindowPtrArray[kMaxWindows + 2];

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

void  main                              (void);
void  doInitManagers                  (void);
void  eventLoop                         (void);
void  doEvents                          (EventRecord *);
void  doMouseDown                           (EventRecord *);
void  doUpdate                          (EventRecord *);
void  doUpdateWindow                    (EventRecord *);
void  doActivate                        (EventRecord *);
void  doActivateWindow                  (WindowPtr,Boolean);
void  doOSEvent                             (EventRecord *);
void  doMenuChoice                      (SInt32);
void  doFileMenu                        (SInt16);
void  doWindowsMenu                         (SInt16);
void  doNewWindow                           (void);
void  doCloseWindow                         (void);
void  doInvalidateScrollBarArea (WindowPtr);
void  doSetStandardState              (WindowPtr);
void  doConcatPStrings                (Str255,Str255);
void  doErrorAlert                    (SInt16);

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ main

void  main(void)
{
    Handle        menubarHdl;
    MenuHandle menuHdl;
    SInt16        a;

    // .......................................................... get nonrelocatable block low in heap for first window structure

    if(!(gPreAllocatedBlockPtr = NewPtr(sizeof(WindowRecord))))
        doErrorAlert(eFailMemory);

    //
..........................................................................................................................................................................
...... initialise managers

    doInitManagers();

    // ...... cause the Appearance-compliant menu bar definition function to be called directly

    RegisterAppearanceClient();

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

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

```
  menuHdl = GetMenuHandle(mApple);
  if(menuHdl == NULL)
    doErrorAlert(eFailMenus);
  else
    AppendResMenu(menuHdl,'DRVR');

  // .................................................................................................................................................. initialize
window pointer array

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

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

  eventLoop();
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doInitManagers

void  doInitManagers(void)
{
  MaxApplZone();
  MoreMasters();
  MoreMasters();
  MoreMasters();
  MoreMasters();

  InitGraf(&qd.thePort);
  InitFonts();
  InitWindows();
  InitMenus();
  TEInit();
  InitDialogs(NULL);

  InitCursor();
  FlushEvents(everyEvent,0);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ eventLoop

void  eventLoop(void)
{
  EventRecord eventStructure;

  gDone = false;

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

    if(gPreAllocatedBlockPtr == NULL)
      if(!(gPreAllocatedBlockPtr = NewPtr(sizeof(WindowRecord))))
        doErrorAlert(eFailMemory);
  }
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doEvents

void  doEvents(EventRecord *eventStrucPtr)
{
  SInt8 charCode;

  switch(eventStrucPtr->what)
  {
    case mouseDown:
      doMouseDown(eventStrucPtr);
      break;

    case keyDown:
    case autoKey:
      charCode = eventStrucPtr->message & charCodeMask;
      if((eventStrucPtr->modifiers & cmdKey) != 0)
        doMenuChoice(MenuEvent(eventStrucPtr));
      break;
```

```
        case updateEvt:
            doUpdate(eventStrucPtr);
            break;

        case activateEvt:
            doActivate(eventStrucPtr);
            break;

        case osEvt:
            doOSEvent(eventStrucPtr);
            HiliteMenu(0);
            break;
    }
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doMouseDown

void  doMouseDown(EventRecord *eventStrucPtr)
{
    WindowPtr    windowPtr;
    SInt16       partCode;
    Rect         growRect;
    SInt32       newSize;

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

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

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

        case inDrag:
            DragWindow(windowPtr,eventStrucPtr->where,&qd.screenBits.bounds);
            break;

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

        case inGrow:
            growRect = qd.screenBits.bounds;
            growRect.top    = 80;
            growRect.left = 160;
            newSize = GrowWindow(windowPtr,eventStrucPtr->where,&growRect);
            if(newSize != 0)
            {
                doInvalidateScrollBarArea(windowPtr);
                SizeWindow(windowPtr,LoWord(newSize),HiWord(newSize),true);
                doInvalidateScrollBarArea(windowPtr);
            }
            break;

        case inZoomIn:
        case inZoomOut:
            if(TrackBox(windowPtr,eventStrucPtr->where,partCode))
            {
                SetPort(windowPtr);
                EraseRect(&windowPtr->portRect);
                ZoomWindow(windowPtr,partCode,false);
            }
            break;
    }
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doUpdate

void  doUpdate(EventRecord *eventStrucPtr)
{
    WindowPtr    windowPtr;

    windowPtr = (WindowPtr) eventStrucPtr->message;
```

```
  BeginUpdate(windowPtr);

  SetPort(windowPtr);
  doUpdateWindow(eventStrucPtr);

  EndUpdate(windowPtr);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doUpdateWindow

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

  windowPtr = (WindowPtr) eventStrucPtr->message;

  EraseRgn(windowPtr->visRgn);

  theRect = windowPtr->portRect;
  theRect.right -= 15;
  theRect.bottom -= 15;

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

  DrawGrowIcon(windowPtr);

  RGBForeColor(&whiteColour);
  TextSize(10);

  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)
{
  WindowPtr  windowPtr;
  Boolean    becomingActive;

  windowPtr = (WindowPtr) eventStrucPtr->message;

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

  doActivateWindow(windowPtr,becomingActive);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doActivateWindow

void  doActivateWindow(WindowPtr windowPtr,Boolean becomingActive)
{
  MenuHandle windowsMenu;
  SInt16     menuItem, a = 1;

  windowsMenu = GetMenuHandle(mWindows);

  while(gWindowPtrArray[a] != windowPtr)
    a++;
  menuItem = a;

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

```
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doOSEvent

void  doOSEvent(EventRecord *eventStrucPtr)
{
    switch((eventStrucPtr->message >> 24) & 0x000000FF)
    {
        case suspendResumeMessage:
            if(gCurrentNumberOfWindows > 0)
            {
                gInBackground = (eventStrucPtr->message & resumeFlag) == 0;
                doActivateWindow(FrontWindow(),!gInBackground);
            }
            break;
    }
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doMenuChoice

void  doMenuChoice(SInt32 menuChoice)
{
    SInt16  menuID, menuItem;
    Str255  itemName;
    SInt16  daDriverRefNum;

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

    if(menuID == 0)
        return;

    switch(menuID)
    {
        case mApple:
            if(menuItem == iAbout)
                SysBeep(10);
            else
            {
                GetMenuItemText(GetMenuHandle(mApple),menuItem,itemName);
                daDriverRefNum = OpenDeskAcc(itemName);
            }
            break;

        case mFile:
            doFileMenu(menuItem);
            break;

        case mWindows:
            doWindowsMenu(menuItem);
            break;
    }

    HiliteMenu(0);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doFileMenu

void  doFileMenu(SInt16 menuItem)
{
    switch(menuItem)
    {
        case iNew:
            doNewWindow();
            break;

        case iClose:
            doCloseWindow();
            break;

        case iQuit:
            gDone = true;
            break;
    }
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doWindowsMenu

void  doWindowsMenu(SInt16 menuItem)
{
```

```
    WindowPtr    windowPtr;

    windowPtr = gWindowPtrArray[menuItem];
    SelectWindow(windowPtr);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doNewWindow

void  doNewWindow(void)
{
    WindowPtr     windowPtr;
    Str255        untitledString;
    Str255        numberAsString;
    MenuHandle windowsMenu;

    if(gCurrentNumberOfWindows == kMaxWindows)
    {
        doErrorAlert(eMaxWindows);
        return;
    }

    if(!(windowPtr = GetNewCWindow(rNewWindow,gPreAllocatedBlockPtr,(WindowPtr)-1)))
        doErrorAlert(eFailWindow);

    gPreAllocatedBlockPtr = NULL;

    GetIndString(untitledString,rStringList,sUntitled);
    NumToString(++gUntitledWindowNumber,numberAsString);
    doConcatPStrings(untitledString,numberAsString);
    SetWTitle(windowPtr,untitledString);

    doSetStandardState(windowPtr);

    ShowWindow(windowPtr);

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

    SetWRefCon(windowPtr,gCurrentNumberOfWindows);

    gCurrentNumberOfWindows ++;
    gWindowPtrArray[gCurrentNumberOfWindows] = windowPtr;

    if(gCurrentNumberOfWindows == 1)
    {
        EnableItem(GetMenu(mFile),iClose);
        EnableItem(GetMenu(mWindows),0);
        DrawMenuBar();
    }
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doCloseWindow

void  doCloseWindow(void)
{
    WindowPtr     windowPtr;
    MenuHandle windowsMenu;
    SInt16        a = 1;

    windowPtr = FrontWindow();
    CloseWindow(windowPtr);
    DisposePtr((Ptr) (WindowPeek) windowPtr);
    gCurrentNumberOfWindows --;

    windowsMenu = GetMenu(mWindows);
    while(gWindowPtrArray[a] != windowPtr)
        a++;
    gWindowPtrArray[a] = NULL;
    DeleteMenuItem(windowsMenu,a);

    for(a=1;a<kMaxWindows+1;a++)
    {
        if(gWindowPtrArray[a] == NULL)
```

```
        {
            gWindowPtrArray[a] = gWindowPtrArray[a+1];
            gWindowPtrArray[a+1] = NULL;
        }
    }

    if(gCurrentNumberOfWindows == 0)
    {
        DisableItem(GetMenu(mFile),iClose);
        DisableItem(GetMenu(mWindows),0);
        DrawMenuBar();
    }
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doInvalidateScrollBarArea

void  doInvalidateScrollBarArea(WindowPtr windowPtr)
{
    Rect tempRect;

    SetPort(windowPtr);

    tempRect = windowPtr->portRect;
    tempRect.left = tempRect.right - 15;
    InvalRect(&tempRect);

    tempRect = windowPtr->portRect;
    tempRect.top = tempRect.bottom - 15;
    InvalRect(&tempRect);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doSetStandardState

void  doSetStandardState(WindowPtr windowPtr)
{
    WindowPeek windowRecPtr;
    WStateData  *winStateDataPtr;
    Rect             tempRect;

    tempRect = qd.screenBits.bounds;
    windowRecPtr = (WindowPeek) windowPtr;
    winStateDataPtr = (WStateData *) *(windowRecPtr->dataHandle);

    SetRect(&(winStateDataPtr->stdState),tempRect.left+40,tempRect.top+60,
                 tempRect.right-40,tempRect.bottom-40);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ 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;
    Str255                          labelText;
    Str255                          narrativeText;
    SInt16                          itemHit;

    paramRec.movable              = false;
    paramRec.helpButton           = false;
    paramRec.filterProc         = NULL;
    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);
   }
   else
   {
      StandardAlert(kAlertStopAlert,labelText,0,&paramRec,&itemHit);
      ExitToShell();
   }
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊
```

Demonstration Program 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, close and size boxes, noting window updating and activation.
- Choose Show Balloons from the Help menu and move the cursor over the hot rectangles in any window.

If an attempt is made to open more than 10 windows, a modal alert box appears. If Show Balloons has been chosen, the user will note that the windows are not updated when overlaying balloons are closed. The reason for this, and the cure, is addressed at Chapter 8 — Dialogs and Alerts.

#define

The first ten #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. MAXLONG is defined as the maximum possible long value. (This will be assigned to WaitNextEvent's sleep parameter.) The (fairly common) macro which follows is required by the application-defined string concatenation function doConcatPStrings.

Global Variables

The global variable gDone, when set to true, causes the main event loop to be exited and the program to terminate. gInBackground relates to foreground/background switching.

gPreAllocatedBlockPtr will be assigned a pointer to a pre-allocated block of memory for a window structure. 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.

gWindowPtrArray[] is central to the matter of maintaining an association between item numbers in the Windows 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 Windows menu item is chosen, the program must be able to locate the window structure 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 pointers for each opened window to the elements of gWindowPtrArray[], starting with gWindowPtrArray[1] and leaving gWindowPtrArray[0] unused. If, for example, six windows are opened in sequence, gWindowPtrArray[1] to gWindowPtrArray[6] are assigned the window pointers for each of those six windows. (At the same time, menu items representing each of those windows are progressively added to the Windows menu.)

If, say, the third window opened is then closed, gWindowPtrArray[3] is set to NULL and the window pointers in gWindowPtrArray[4] to gWindowPtrArray[6] are moved down in the array to occupy gWindowPtrArray[3] to

gWindowPtrArray[5]. Since the Windows 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 pointer is assigned and the number of the menu item for that window.

main

The first line in the main function requires some explanation. When a window is created, its window structure is contained in a nonrelocatable block of memory. Any program that allows the user to open many windows at any time during program execution must have a strategy for allocating all window structures as low in the heap as possible, since nonrelocatable blocks scattered within the heap contribute to memory fragmentation and impede effective heap compaction by the Memory Manager.

The best times to allocate nonrelocatable blocks so as to ensure that they are located as low in the heap as possible are:

- At the beginning of the program (just before the system software managers are initialised).

- At the bottom of the event loop just after all events have been handled to completion. At this time, the heap is as empty as it will ever be.

This program adopts that strategy. The first line in main() pre-allocates a nonrelocatable block which will later be used by the window structure of the first window to be created. The pointer returned by the first call to GetNewCWindow, which will be copied to gWindowPtrArray[1], will point to this block. gPreAllocatedBlockPtr will then be set to NULL. At the bottom of the event loop, gPreAllocatedBlockPtr will be checked. If it contains NULL, the pre-allocated block must now be occupied by a window structure, in which circumstance a new block will be allocated in preparation for the next window to be opened.

If the call to NewPtr fails, the following line invokes an Alert box.

The system software managers are then initialised.

The call to RegisterAppearanceClient means that the new Appearance-compliant menu bar definition function (resource ID 63) will be used regardless of whether system-wide Appearance is selected on or off in the Appearance control panel.

The next block sets up the menus. Note that error handling involving the invocation of alert boxes is introduced in this program. If an error occurs, the application-defined function doErrorAlert will display an alert box advising of the nature of the error before terminating the program.

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

doInitManagers

Note that MoreMasters must be called four times to provide sufficient master pointers for this particular program. (The requirement for two calls to MoreMasters was determined by watching master pointer usage during program execution using ZoneRanger.)

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 MAXLONG, which is defined as the maximum possible long value.)

At the bottom of the event loop, a new nonrelocatable block is allocated in preparation for the next window to be opened if the global variable gPreAllocatedBlockPtr contains NULL.

doEvents

doEvents switches according to the event type received.

mouseDown, upDate, activateEvt and osEvt events are of significance to the windows aspects of this demonstration. To that extent, keyDown events are significant only with regard to Windows menu keyboard equivalents.

Note that the call to HiliteMenu at the second last line is required to unhighlight the Apple menu title when the application is brought to the foreground again following a period of dalliance with an item in the Apple menu (other than the About… item).

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. screenBits is a QuickDraw global variable of type BitMap. The bounds field of screenBits is a Rect containing the coordinates of a rectangle which encloses the main screen.

The inGoAway case results in a call to TrackGoAway, which retains control until the user releases the mouse button. If the pointer was still within the go away box when the button was released, the application-defined function doCloseWindow is called.

The inGrow case first sets up the Rect used in the third parameter of the GrowWindow call which, in turn, will limit the maximum size to which the window can be resized. The top, left, bottom and right fields must contain, respectively, the minimum vertical, the minimum horizontal, the maximum vertical, and the maximum horizontal measurements. At the first line, this Rect is set to the boundaries of the screen, which is a reasonable way to get reasonable values into the bottom and right fields. The top and left fields, however, need to be manually set to some reasonable values (the two lines before the GrowWindow call).

GrowWindow retains control until the user releases the mouse button, at which time the Rect variable newSize will contain the new window size coordinates. (Note that GrowWindow does not redraw the window in this size.) The SizeWindow call then redraws the window frame and title and, where window height and/or width has been increased, adds the newly-exposed areas to the update region.

The call SizeWindow is bracketed by two calls to the application-defined 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 area less the areas normally occupied by scroll bars and, secondly, to use DrawGrowIcon to draw the draw scroll bar delimiting lines. (This is the usual practice for windows with a size box but no scroll bars.)

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 application-defined 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, 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.)

The inZoomIn and inZoomOut cases result in a call to TrackBox, which takes control until the user releases the mouse button. If the mouse button is released while the pointer is still within the zoom box, the current colour graphics port is set to that associated with the active window. the content region is erased, and ZoomWindow is called to redraw the window frame and title in the new zoomed state, which will be either the user state or the standard state. (ZoomWindow knows which way to go because the Window Manager keeps track of the current state, which is contained in the partCode variable returned by FindWindow and passed to ZoomWindow as its second parameter.)

doUpdate

doUpdate attends to basic window updating. 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 colour graphics port is then set before the application-defined function doUpdateWindow is called to redraw the content region. The EndUpdate call restores the window's true visible region.

doUpdateWindow

doUpdateWindow is concerned with redrawing the window's contents less the scroll bar areas. Following the extraction of the window pointer from the message field of the event structure, EraseRgn is called for reasons explained at doMouseDown, above.

A Rect is then assigned the coordinates of that window's colour graphics port portRect field, 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 value in the window structure's refCon field. As will be seen, whenever a new window is opened, a value between 1 and kMaxWindows is assigned to this field. 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. 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.

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 application-defined function doActivateWindow.

doActivateWindow

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

The first step in the function doActivateWindow is to associate the received WindowPtr with its item number in the Windows 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.

Note that, if you required scroll bar delimiting lines to be drawn in the window, you would call DrawGrowIcon within this function. (In Appearance-compliant windows, DrawGrowIcon does not draw the grow icon itself.)

doOSEvent

doOSEvent handles operating system events. In this demonstration, action is taken only in the case of suspend and resume events (first line) and then only if at least one window is open (second line).

The action taken is to set the global variable gInBackground to true or false depending on whether the event is a suspend event or a resume event. In the case of a suspend event, window deactivation tasks needs to be performed. In the case of a resume event, activation tasks need to be attended to. Accordingly, doActivateWindow is called with the second parameter set to true for a resume and to false for a suspend.

doMenuChoice

doMenuChoice switches according to the menu choices of the user.

doFileMenu

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

doWindowsMenu

doWindowsMenu takes the item number of the selected Windows menu item and, since this equates to the number of the array element in which the associated window pointer is stored, extracts the window pointer 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 Alert is called up via the application-defined doErrorAlert function (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. The second parameter of the GetNewCWindow call is a pointer to the pre-allocated block of memory allocated earlier in the program, and 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 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.

If the window was successfully opened, gPreAllocatedBlockPtr is set to NULL so that a new pre-allocated block will be created at the bottom of the event loop in preparation for the next window to be opened.

The next four lines insert the number of the window into the title bar (for example, "Untitled 1" for the first window opened). GetIndString retrieves the string "Untitled " from the 'STR#' resource. Within the first parameter of the NumToString call, the global variable which keeps track of the numbers for the title bar is incremented before NumToString converts that number to a Pascal string. The next line concatenates this string to the "Untitled " string. The SetWTitle call then changes the window's title and redraws the title bar.

The window's standard state is set by the call to the application-defined function doSetStandardState. (If the standard state is not set programmatically like this, the system will automatically set it 3 pixels inside the screen's gray region boundary.)

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 Windows menu. Note that the Command-key equivalent is only added for the first nine opened windows.)

The SetWRefCon call assigns a value to the window structure's reference constant (refCon) field. 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 pointer array is assigned the window pointer of the newly opened window.

The last block enables the Windows 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 pointer to the frontmost window is retrieved and that window is closed by a call to CloseWindow. CloseWindow, rather than DisposeWindow, must be used where storage for the window structure was allocated manually, that is, where the second parameter in the GetNewCWindow call was not NULL. Because CloseWindow is used, the following call to DisposePtr is necessary to dispose of the non-relocatable block occupied by the window structure. With the window closed, the global variable which keeps track of the number of windows currently open is decremented.

The next block deletes the associated menu item from the windows menu. At the first four lines, the array element in which the WindowPtr 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 are which would be occupied by scroll bars. The function simply retrieves the coordinates of the content area into a 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.

doSetStandardState

The function doSetStandardState sets the window's standard state. First the coordinates of the screen boundary are placed in a Rect. Then, in the next two lines, the handle in the window structure's dataHandle field is dereferenced to a pointer and cast to a pointer to a WStateData structure. At the last two lines, this pointer is then used in the call to SetRect, which sets the required top, left, bottom, and right values in the stdState field of the window's WStateData structure.

doConcatPStrings

The function doConcatPStrings concatenates two Pascal strings.

doErrorAlert

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.

The creation of Alerts using the StandardAlert function is addressed at Chapter 8 — Dialogs and Alerts.

Handling Horizontal and Vertical Zoom Boxes

The following details those changes that would be required to handle a vertical zoom box in the windows of the Windows demonstration program. It assumes that, when the vertical zoom box is clicked, the window position is to remain unchanged and the window size is to change only in the vertical direction.

Firstly, using Resorcerer, change the window definition ID in the 'WIND' resource editing window to kWindowVertZoomDocumentProc (1026). (This will simply change the appearance of the zoom box.) Then make the following changes to the source code.

Delete the function prototype for doSetStandardState and insert this new function prototype:

```
void  doSetUserAndStandardState  (WindowPtr);
```

In the function doNewWindow, change the line doSetStandardState(windowPtr); to:

```
doSetUserAndStandardState(WindowPtr);
```

In the function doMouseDown, change the inDrag case as follows:

```
case inDrag:
   DragWindow(windowPtr,eventStrucPtr->where,&qd.screenBits.bounds);
   doSetUserAndStandardState(windowPtr); // New call to set the user and standard state to
                                                   // reflect the new window location after the drag.
   break;
```

Replace the function doSetStandardState with this new function

```
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doSetUserAndStandardState

void  doSetUserAndStandardState(WindowPtr windowPtr)
{
   WindowPeek windowRecPtr;
   WStateData  *winStateDataPtr;
   Rect            tempRect;
   Point           topLeft;

   // Convert the top left corner of the window's content region to global coordinates.
   // (This gets the current location of the window.)

   SetPort(windowPtr);
   topLeft.v = 0;
   topLeft.h = 0;
   LocalToGlobal(&topLeft);

   // Get a pointer to the window's WStateData structure

   windowRecPtr = (WindowPeek) windowPtr;
   winStateDataPtr = (WStateData *) *(windowRecPtr->dataHandle);

   // Set the standard state to equate with the current position of the window and the
   // required size of the window in the zoomed-out state (100 pixels higher than in the
   // zoomed-in state)..

   SetRect(&(winStateDataPtr->stdState),topLeft.h,topLeft.v,topLeft.h + 460,
               topLeft.v + 300 + 100);

   // Set the user state to equate with the current position of the window and the required
   // size of the window in the zoomed-in state.

   SetRect(&(winStateDataPtr->userState),topLeft.h,topLeft.v,topLeft.h + 460,
               topLeft.v + 300);
}
```