4

Version 1.1

WINDOWS

Includes Demonstration Program WindowsPascal

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 defines and supports a set of standard windows and provides routines for managing them, itself depends on QuickDraw. QuickDraw supports drawing into **graphics ports**, which are individual and complete drawing environments with independent coordinate systems. Each window represents a 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.

Standard Window Elements

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

- The **title bar** displays the name of the window and indicates whether it is active or inactive. If the system font is in the Roman script system, the title bar is 20 pixels high. 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. The user expects to be able to move the window by dragging its title bar.

- The **close box** (or **go-away box**) offers the user a quick way to close a window.

- The **zoom box** offers the user a quick way to choose between two different window sizes, one established by the user and one by the application.

- The **size box** lets the user change the size of the window.

Other elements of a window are as follows:

- The **content region** is the part of the window in which your application displays the contents of the document, the size box and the window's controls.

- The window **frame** is the part of the window drawn automatically by the Window Manager, that is, the title bar (including the close box and zoom box) and the window's outline.

- The **structure region** is the entire screen area occupied by a window, including the frame and content region.

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, identified by the "racing stripes" in its title bar. The active window is the target of all keyboard activity and only the active window interacts with the user. Scroll bars and highlighting should be displayed only in the active window.

When the user activates one of your application's windows, the Window Manager highlights the window frame and title bar. Your application must reinstate the appearance of the rest of the window to its state prior to the deactivation, activating the controls, 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 frame, removing the highlighting from the title bar and hiding the close and zoom boxes. Your application must hide the controls and the size box, remove highlighting from application-controlled elements, and so on.

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

Types of Windows

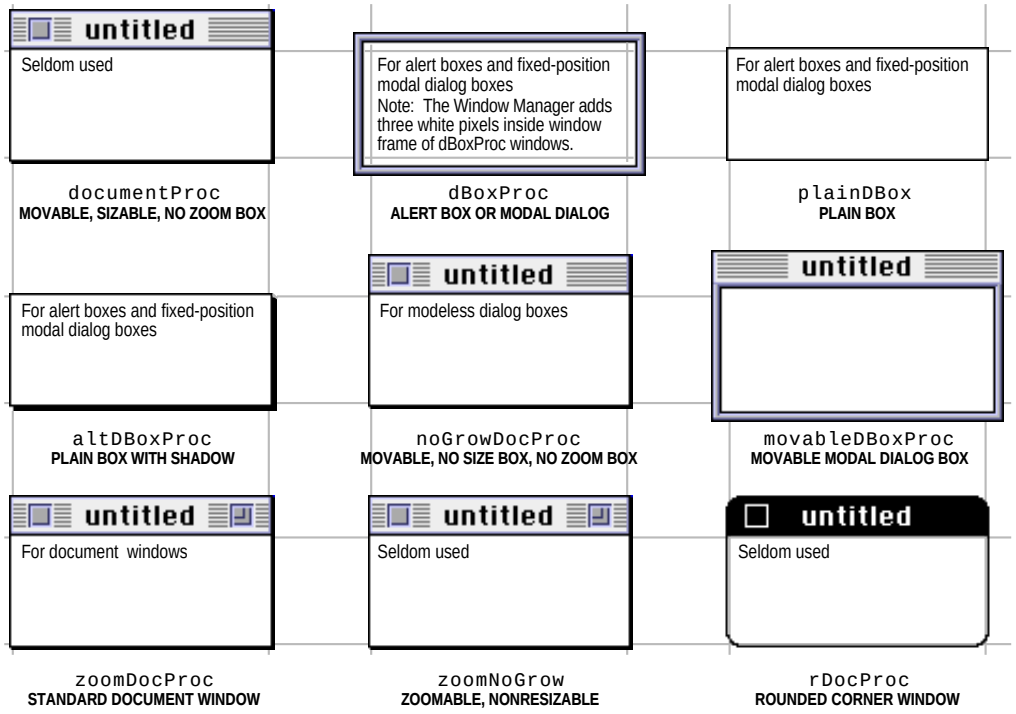

FIG 1 - TYPES OF WINDOWS

The Window Manager defines nine basic window **types**. These nine basic types are often referred to by the constant used in `'WIND'` resources, and by certain Window Manager routines, to specify the type of

window required. That constant determines both the visual appearance of the window and its behaviour.

Fig 1 shows the nine available basic window types, the constants which represent those types, and the general usage of each type. Note that all of these windows are shown at the same size (120 pixels wide by 50 pixels high).

Window Definition IDs

The nine constants shown at Fig 1 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 routines call a window's window definition function when they need to perform certain window-related actions, such as drawing or resizing a window's frame. The definition function draws the window's frame, draws the close box and window title (if any), determines which region the cursor is in within the window, calculates the window's content and structure regions, draws the window's zoom box (if any), draws the window's size box (if any) and performs any special initialisation or disposal tasks.

- **Variation Code.** A single window definition function can support up to 16 different window types. The window definition function defines a **variation code**, an integer from 0 to 15, for each window type it supports.

Two window definition functions are associated with the nine basic window types shown at Fig 1. These reside, in two resources of type `'WDEF'`, in the System file in the System Folder. The resource IDs are 0 and 1. The window definition ID is derived by multiplying the `'WDEF'` resource ID by 16 and adding the variation code to the result, as is shown in the following:

`'WDEF'` Resource ID	Variation Code	Window Definition ID (Decimal)	Window Definition ID (Constant)
0	0	0 * 16 + 0 = 0	documentProc
0	1	0 * 16 + 1 = 1	dBoxProc
0	2	0 * 16 + 2 = 2	plainDBox
0	3	0 * 16 + 3 = 3	altDBoxProc
0	4	0 * 16 + 4 = 4	noGrowDocProc
0	5	0 * 16 + 5 = 5	movableDBoxProc
0	8	0 * 16 + 8 = 8	zoomDocProc
0	12	0 * 16 + 12 = 12	zoomNoGrow
1	0	1 * 16 + 0 = 16	rDocProc

Window Type Usage

Window Types For Documents. A `zoomDocProc` 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 the Window Manager does not draw the grow icon in the size box. (Your application must call `DrawGrowIcon` for that purpose.) And, of course, when the related document contains more data that will fit in the window, you must add scroll bars.

Window Types For Alert Boxes and Fixed-Position Modal Dialog Boxes. Alert boxes and dialog boxes are merely special-purpose windows. Since they require no window manipulation elements and remain on the screen as the active window until the Dialog Manager or your application removes them, alert boxes and fixed-position modal dialog boxes use the window types `dBoxProc`, `plainDBox` and `altDBoxProc`, most typically `dBoxProc`.[1]

[1]You can handle all alert boxes and most modal dialog boxes through the Dialog Manager, which itself calls the Window Manager. You supply the Dialog Manager with a list of items in your alert boxes and dialog boxes, and the Dialog Manager displays the windows, tells you which items the user is manipulating, and disposes of the windows when the user is done.

Window Types For Movable Modal Dialog Boxes. Movable modal dialog boxes are used when you want to allow the user to move a modal dialog box window in order, for example, to view text obscured by that window. Like the fixed-position modal dialog box, the movable modal dialog box remains active until the user completes the dialog. The `movableDBoxProc` type is used for movable modal dialog boxes.[2]

Window Types For Modeless Dialog Boxes. Modeless dialog boxes allow the user to perform other tasks without first dismissing the dialog box. Modeless dialogs should thus be used in favour of modal dialogs wherever possible. User interface guidelines require that the `noGrowDocProc` window type, which can be moved or closed but not resized or zoomed, be used for modeless dialog boxes.[3]

The creation and handling of alert and dialog boxes is addressed in detail at Chapter 6— Dialogs and Alerts.

Other Window Definitions

If you need a window with unusual characteristics, you can write your own window definition function. On the other hand, you may find that one or other of the additional window definitions provided in the System file in the System Folder will suit your requirements. For example, the rather unusual windows which appear when you choose **Macintosh Guide** from the Help menu utilize the `'WDEF'` resource with ID 124 in the System file. Fig 2 shows the twelve window types available using this particular window definition function.[4]

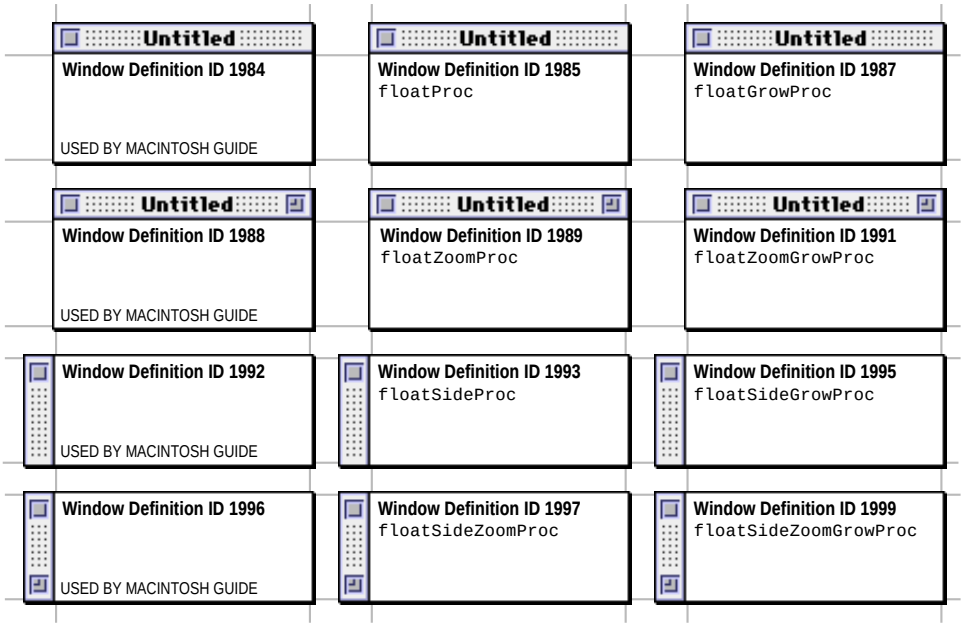

FIG 2 - OTHER WINDOW TYPES AVAILABLE USING 'WDEF' RESOURCE WITH ID 124 FROM THE SYSTEM FILE IN THE SYSTEM FOLDER

The following shows the window definition ID used to specify each of these window types, together with its derivation.

`'WDEF'` Resource ID	Variation Code	Window Definition ID (Decimal)	Window Definition ID (Constant)
124	0	124 * 16 + 0 = 1984	
124	1	124 * 16 + 1 = 1985	floatProc
124	3	124 * 16 + 3 = 1987	floatGrowProc

[2]Although the Dialog Manager will help handle events in movable modal dialog boxes, your application must handle the dragging of the movable modal dialog box window.

[3]The Dialog Manager helps handle events in modeless dialog boxes,; however, your application must handle the window manipulation events for modeless dialog boxes, just as it handles such events in document windows.

[4]Macintosh Guide and this particular `'WDEF'` resource were introduced with System 7.5.

```
124          4        124 * 16 + 4  = 1988
124          5        124 * 16 + 5  = 1989    floatZoomProc
124          7        124 * 16 + 7  = 1991    floatZoomGrowProc
124          8        124 * 16 + 8  = 1992
124          9        124 * 16 + 9  = 1993    floatSideProc
124         11        124 * 16 + 11 = 1995    floatSideGrowProc
124         12        124 * 16 + 12 = 1996
124         13        124 * 16 + 13 = 1997    floatSideZoomProc
124         15        124 * 16 + 15 = 1999    floatSideZoomGrowProc
```

Window Regions

The Window Manager recognises the following special-purpose **window regions**, which are defined by either the Window Manager or the window definition function:

- The **drag region**, **close region**, **size region** and **zoom region**.
- The **structure region**, which is the entire area occupied by the window, including the window outline, title bar and content region.
- The **content region**, to which the drawing region of a graphics port is confined.
- The **update region**, a dynamic region which accumulates all areas of a window's content region which need updating.

Controls and Control Lists

Windows may contain **controls**. The most common control in a window is the **scroll bar** (see Fig 3), 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 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.

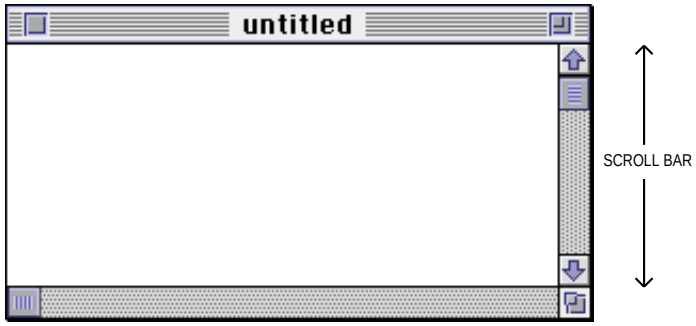

FIG 3 - SCROLL BARS

Windows on the Desktop

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 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 global variable named `GrayRgn`. You can retrieve a pointer to the gray region with the Window Manager function `GetGrayRgn`.

Graphics Ports

Each window represents a QuickDraw **graphics port**, which is a drawing environment with its own coordinate system. The Window Manager creates a 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. In QuickDraw routines, 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. (See Fig 4.)

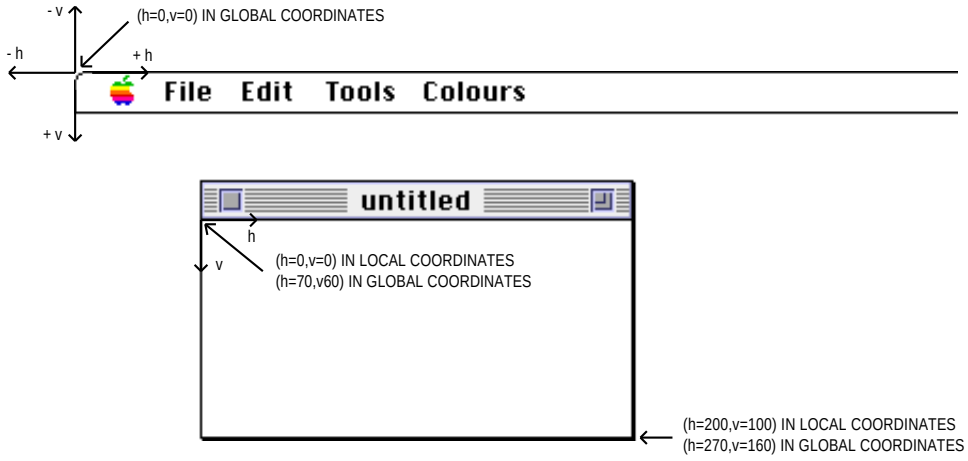

FIG 4 - A WINDOW'S LOCAL AND GLOBAL COORDINATE SYSTEMS

When QuickDraw creates a new 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 graphics port data structure, in the `bounds` field of a structure called a **bitmap** in Basic QuickDraw and a **pixel map** in Color QuickDraw.

The graphics port data structure also includes a field called `portRect`, which defines the rectangle to be used for drawing. In a 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 4 illustrates the local and global coordinate systems for a window which is 100 pixels high by 200 pixels wide, and which is placed with its content region 70 pixels down and 60 pixels to the right of the upper left corner of the screen.

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

Window Records

The Window Manager stores information about a window in a **window record** or a **colour window record**. A colour window record is defined by the data type `CWindowRecord` and a window record is

defined by the data type `WindowRecord`. The only difference between a window record and a colour window record is that the `port` field is a `GrafPort` rather than a `CGrafPort`:

```
TYPE
  CWindowPeek = ^CWindowRecord;
  CWindowRecord = RECORD
    port:           CGrafPort;
    windowKind:     integer;
    visible:        boolean;
    hilited:        boolean;
    goAwayFlag:     boolean;
    spareFlag:      boolean;
    strucRgn:       RgnHandle;
    contRgn:        RgnHandle;
    updateRgn:      RgnHandle;
    windowDefProc:  Handle;
    dataHandle:     Handle;
    titleHandle:    StringHandle;
    titleWidth:     integer;
    controlList:    ControlRef;
    nextWindow:     WindowPeek;
    windowPic:      PicHandle;
    refCon:         longint;
    END;

  CGrafPtr = ^CGrafPort;
  CWindowPtr = CGrafPtr;


  WindowPeek = ^WindowRecord;
  WindowRecord = RECORD
    port:           GrafPort;
    ...
    The remainder of a WindowRecord is identical to a CWindowRecord.
    END;

  GrafPtr = ^GrafPort;
  WindowPtr = GrafPtr;
```

It is important to note that the graphics port is the first field of both records and that the data types `WindowPtr` and `CWindowPtr` are defined as pointers to the graphics port, not to the window record. Fields in the window record are accessed using `WindowPeek` and `CWindowPeek`, which are pointers to a window record. (`WindowPeek` and `CWindowPeek` are rarely used, however, since you usually do not need to access or directly modify fields in a window record. The Window Manager automatically updates the window record when you make changes to a window, and supplies routines for changing and reading some parts of the window record.)

Windows should be created with a window record only when Color QuickDraw[5] is not available, otherwise they should be created with a colour window record.

The close box, drag region, zoom box, and size box are not included in the window record. The window definition function determines the location of those particular regions.

Compatibility

For compatibility, the `WindowPtr` and `WindowPeek` data types can point to either a window record or a colour window record. In addition, all non-colour Window Manager routines work with the colour window record by accepting a `CWindowPtr` as well as a `WindowPtr` as the graphics port pointer parameter.

Colour Windows

The Window Manager has supported colour windows since the introduction of Color QuickDraw. Colour windows are displayed in colour graphics ports.

[5]Color QuickDraw is not available on black-and-white Macintoshes, such as the Macintosh Classic.

Whether or not your application uses colour explicitly, and whether or not a colour monitor is installed, your application should work successfully with colour windows whenever Color QuickDraw is available. On a monitor that is set to display 4-bit colour (16 colours) or greater, the Window Manager automatically displays the window title and parts of the frame and controls in colour (or gray scale, depending on the capabilities of the monitor). On a monitor set to display 1-bit colour, the Window Manager draws the window title, frame and controls in black and white. Once you have created a window, you can use the window record and window pointer for a colour window interchangeably with the window record and window pointer for a black-and-white window.

Various elements of a window's colours are controlled by the **window colour table**, which contains a series of part codes for different window elements and the RGB (red-green-blue) values associated with each part. Your application typically uses the default colour table, basically because users can change the window display colours for the entire desktop at will using the Color control panel. If your application explicitly controls the colours used in a window, however, you can define your own window colour tables in a `'wctb'` resource with the same resource ID as the window's `'WIND'` resource.[6] The Window Manager will then create a window colour table from the resource when it creates the window record, maintaining its own linked list, using **auxiliary window records**, which associate your application's windows with their corresponding window colour tables.

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 Windows

You typically create windows from resources of type `'WIND'`. Alert box windows and dialog box windows use `'ALRT'`, `'DLOG'` and item list (`'DITL'`) resources. Most windows contain scroll bars, which are defined in `'CNTL'` resources.

Defining a `'WIND'` Resource

You typically define a `'WIND'` resource for each type of window your application creates. The following is an example of a `'WIND'` resource in Rez input format:

```
#define rDocWindow 128

resource 'WIND' (rDocWindow, preload, purgeable)
{
  {64,60,314,460},  /* Initial window size and location. */
  zoomDocProc,      /* Window definition ID (definition function + variation code). */
  invisible,        /* Window is initially invisible. */
  goAway,           /* Window has a close box. */
  $0,               /* Reference constant. */
  "untitled",       /* Window title. */
  staggerParentWindowScreen   /* Optional positioning specification. */
};
```

Window Size and Location. The four numbers in the first element specify the upper-left and lower-right corners, in global coordinates, of a rectangle which defines the initial size and placement of the

[6] If you use ResEdit to create your `'WIND'` resource, the `'wctb'` resource will be created automatically if you specify custom colours within the `'WIND'` resource editor.

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 second element is the window definition ID, which establishes the type of window.

Visible/Invisible. The third element specifies whether the window is to be visible or invisible when the Window Manager creates it.

Close Box/No Close Box. If the fourth element specifies a close box, the field has no effect if the second field specifies a type which does not support close boxes.

Reference Constant. The fifth element is a reference constant in which your application can store whatever data it needs. When it builds a new window record, the Window Manager stores the value specified here in the window record's `refCon` field. (You can also put a placeholder here (such as `$0`) and then set the `refCon` field yourself by calling `SetRefCon`.)

Positioning. The optional seventh element specifies a positioning rule, which overrides the first element. In the window resource for a document window, you typically specify the positioning constant `staggerMainScreen`. The full range of positioning constants is as follows:

Constant	Value	Meaning
`noAutoCenter`	`0x0000`	Use initial location.
`centerMainScreen`	`0x280A`	Centre on main screen.
`alertPositionMainScreen`	`0x300A`	Place in alert position on main screen.
`staggerMainScreen`	`0x380A`	Stagger on main screen.
`centerParentWindow`	`0xA80A`	Center on parent window.
`alertPositionParentWindow`	`0xB00A`	Place in alert position on parent window
`staggerParentWindow`	`0xB80A`	Stagger relative to parent window.
`centerParentWindowScreen`	`0x680A`	Center on parent window screen.
`alertPositionparentWindowScreen`	`0x700A`	Alert position on parent window screen.
`staggerParentWindowScreen`	`0x780A`	Stagger on parent window screen.

The positioning constants represent a convenient method for ensuring correct window placement when the user creates a new document window.

Creating the Window From the `'WIND'` Resource

`GetNewCWindow` and `GetNewWindow` are used to create a window from a `'WIND'` resource.[7] `GetNewCWindow` should be used to create colour windows whenever Color QuickDraw is available, whether or not a colour monitor is currently installed.

You can allow `GetNewCWindow` and `GetNewWindow` to themselves allocate memory for your window record. However, memory fragmentation effects will be minimised by allocating the memory yourself from a block allocated for such purposes during your application's initialisation routine, and then passing the pointer to `GetNewCWindow`.[8]

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 5 — Controls).

---

[7] Two additional routines (`NewWindow` and `NewCWindow`) may be used to create windows without the use of a `'WIND'` resource. The parameters for these routines allow you to specify the bounding rectangle, title, visibility, definition ID, etc.

[8] However, note that, at some point in the future development of the system software, data structures such as window records will no longer be created in your application's address space. It will then be necessary to invariably allow the system to itself allocate the storage for the window record (by passing `NULL` in the second parameter of the `GetNewCWindow`/`GetNewWindow` call).

Window Visibility

If the `'WIND'` resource specifies that the new resource 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 then invoke its own procedure 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. Typically, this is the gray area of the screen minus three pixels all round.

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

```
type
  WStateData = record
    userState: Rect;  {user state}
    stdState: Rect;   {standard state}
    end;

  WStateDataPtr = ^WStateData;
  WStateDataHandle = ^WStateDataPtr;
```

Returning to the matter of saving the user state and the current state of the window, 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:

```
type
  windowState = record
    userStateRect: Rect;  {User state rectangle.}
    ZoomState: boolean;   {Window state: true = standard state, false = user state.}
    end;
```

```
    windowStatePtr = ^windowState;
    windowStateHdl = ^windowStatePtr;
```

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.[9]

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, the Window Manager to draw the size box, 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 drawing and updating of the content region by assigning a handle to the picture in the `windowPic` field of the window record.

Providing Balloon Help

The system software provides help balloons for the window frame (the title bar, zoom box and close box) of a window created with one of the standard definition functions. You should provide help balloons for your window content region, that is, the size box, controls and data area.

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 window record 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 record containing data relating to window management.

As an example, the `refCon` field could hold a number representing the type of dialog box (in the case of modeless or movable modal dialog boxes) or a handle to an application-defined **document record** (in the case of document windows). The document record might typically hold a handle to the text being edited, handles to the scroll bars, a file reference number and a file system specification for the document's file, plus a flag indicating whether data has changed since the last save, as shown in this example application-defined document record:

```
type
  DocRecord = record
    editRec: TEHandle;
    vScrollBar: ControlHandle;
    hScrollbar: Controlhandle;
    fileRefNum: short;
    fileFSSpec: FSSpec;
    windowDirty: boolean;
    end;

  DocRecordPtr = ^DocRecord;
  DocRecordHdl = ^DocRecordPtr;
```

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.

---

[9]The demonstration program MoreResourcesPascal.p at Chapter 15 — More on Resources shows how to save the window state to the resource fork of a document file.

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 window, your response depends on what type of window is active:

- **Movable Modal Dialog Box.** If the active window is a movable modal dialog box, you should sound the system alert and take no other action. (Note that this is not necessary if the dialog box is being handled by the `ModalDialog` procedure, since in that case the Dialog Manager does not pass the event to your application but sounds the system alert itself. This is also the case if the active window is an alert or modal dialog box.)

- **Document Window or Modeless Dialog Box.** 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.

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

Preamble

The Window Manager maintains an update region, which represents the parts of your content region which have been affected by changes to the desktop. The Event Manager periodically scans the update regions of all windows on the desktop. If it finds one whose update region is not empty, it generates an update event for that window. Your application can receive update events when it is in either the foreground or, provided the application's `'SIZE'` resource so specifies, the background.

When your application receives an update event, it should redraw as much of the content area as is necessary.

Updating Strategies. As the user makes changes to a document, your application must update both the document data and the document display. You can use one of two strategies to update the display:

- If the application does not require rapid scrolling or rapid response, you add changed areas of the content region to the window's update region. The Event Manager then sends your application an update event, and your application invokes its standard update procedure.

- For continuous scrolling and a faster response time, you can draw directly into the content area.

In either case, your application ultimately draws in the graphics port associated with the window.

Manipulating the Update Region. Your application can force or suppress update events by manipulating the update region. It usually does this, for example, when the user resizes a window containing scroll bars. If the user enlarges the window, the Window Manager adds the newly exposed areas to the update region but does not add the area formerly occupied by the scroll bars. Therefore, before calling `SizeWindow` to resize the window, your application must call `InValRect` to add the two areas formerly occupied by the scroll bars to the update region. You can also remove an area from the update region when appropriate, since limiting the size of that region decreases the time spent redrawing. For example, an unaffected text area could be removed from the update region of a window that is being resized.

The Update Process

When your application redraws the content region in response to an update event, 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 graphics port. The visible region is that part of a 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 graphics port data structure.)

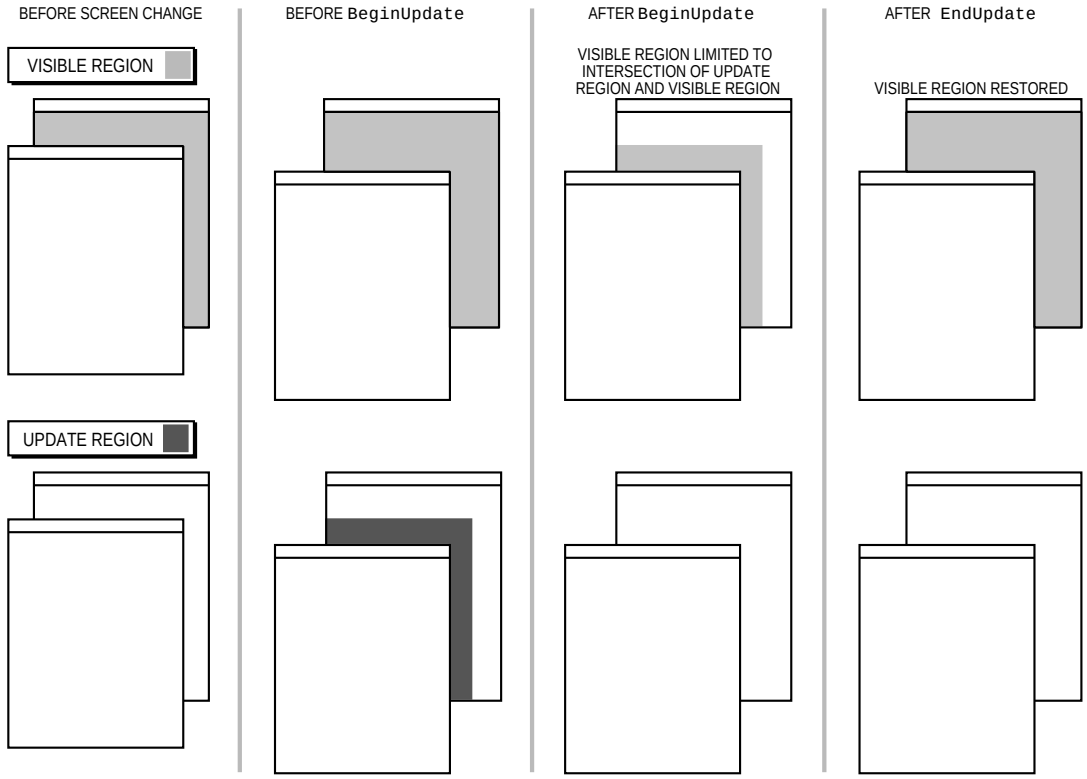

FIG 5 - EFFECTS OF BeginUpdate AND EndUpdate ON VISIBLE AND UPDATE REGIONS

In response to an update event, your application should call `BeginUpdate`, draw the window's contents and then call `EndUpdate`. As shown at Fig 5, `BeginUpdate` limits the visible region to the intersection of the visible region and the update region. Because QuickDraw limits drawing to this modified visible region, only those parts of the window which actually need updating are drawn. `BeginUpdate` also clears the update region.

After your application has updated the window, `EndUpdate` should be called to restore the visible region of the graphics port to the full visible region.[10]

[10]The reason for this update region/visible region swapping is that QuickDraw knows about visible regions but has no knowledge of the existence of update regions. QuickDraw needs something it can work with.

Type-Dependent Update Procedures

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 completed. If the window is a dialog box, an application-defined dialog updating function 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 previous 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 size box and remove any highlighting of selections. When your application receives the event for the newly activated window, it should draw the controls and size box 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 some other window. It should then perform the appropriate activation / deactivation actions. The function does not need to check for alert boxes and 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

The zoom box allows the user to alternate quickly between two window positions and sizes: 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 record whose handle appears in the `dataHandle` field of the window record. The Window Manager sets the initial values of the `userState` and `stdState` fields when it fills in the window record 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.

Resizing 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 gray 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 resizing windows should first save the current view rectangle. `SizeWindow` should then be called to draw the window in its new size. The scroll bars and window contents should then be adjusted to the new size and the content region of the window invalidated with a call to `InvalRect`. The intersection of the old view rectangle and the new view rectangle should then be calculated, this area being used to re-validate unchanged portions of the window, that is, to remove them from the update region. That way, only the changed parts of the content area will be redrawn when the application receives its next update event.

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 14 — 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 record. `DisposeWindow` should be used if you allowed the system to allocate storage for the window record, that is, if you passed `NULL` as the `wStorage` parameter in the `NewWindow`, `GetNewWindow`, `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 record. `CloseWindow` should be used when you have allocated storage for the window record manually, that is, if you created a nonrelocatable block for the window record and passed the pointer as the `wStorage` parameter in the `NewWindow`, `GetNewWindow`, `NewCWindow`, or `GetNewCWindow` call. In this case, the nonrelocatable block containing the window record 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.

Main Window Manager Constants, Data Types and Routines

Constants

Window Types

```
documentProc          = 0
dBoxProc              = 1
plainDBox             = 2
altDBoxProc           = 3
noGrowDocProc         = 4
movableDBoxProc       = 5
zoomDocProc           = 8
zoomNoGrow            = 12
rDocProc              = 16
floatProc             = 1985
floatGrowProc         = 1987
floatZoomProc         = 1989
```

```
floatZoomGrowProc      = 1991
floatSideProc          = 1993
floatSideGrowProc      = 1995
floatSideZoomProc      = 1997
floatSideZoomGrowProc  = 1999
```

Window Kind

```
dialogKind             = 2
userKind               = 8
```

Part Codes Returned by `FindWindow`

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

Data Types

```
type
  WindowRef = WindowPtr;
```

Colour Window Record

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

CWindowPeek = ^CWindowRecord;
```

Window Record

```
WindowRecord = record
  port: GrafPort;
            ...
  The remainder of a WindowRecord is identical to a CWindowRecord.
  end;

WindowPeek = ^WindowRecord;
```

State Data Record

```
WStateData = record
  userState:     Rect;           {user state}
  stdState:      Rect;           {standard state}
  end;

WStateDataPtr = ^WStateData;
WStateDataHandle = ^WStateDataPtr;
```

Auxiliary Window Record

```
AuxWinRec = record
  awNext:      AuxWinHandle;  {handle to next AuxWinRec}
  awOwner:     WindowRef;     {ptr to window}
  awCTable:    CTabHandle;    {color table for this window}
  reserved:    UInt32;
  awFlags:     longint;       {reserved for expansion}
  awReserved:  CTabHandle;    {reserved for expansion}
  awRefCon:    longint;       {user constant}
  end;

AuxWinPtr = ^AuxWinRec;
AuxWinHandle = ^AuxWinPtr;
```

Routines

Initializing the Window Manager

```
procedure  InitWindows;
```

Creating Windows

```
function   GetNewCWindow(windowID: integer; wStorage: UNIV Ptr;
           behind: WindowRef): WindowRef;
function   GetNewWindow(windowID: integer; wStorage: UNIV Ptr; behind: WindowRef): WindowRef;
function   NewCWindow(wStorage: UNIV Ptr; var boundsRect: Rect; title: ConstStr255Param;
           visible: boolean; procID: integer; behind: WindowRef; goAwayFlag: boolean;
           refCon: longint): WindowRef;
function   NewWindow(wStorage: UNIV Ptr; var boundsRect: Rect; title: ConstStr255Param;
           visible: boolean; theProc: integer; behind: WindowRef; goAwayFlag: boolean;
           refCon: longint): WindowRef;
```

Naming Windows

```
function   GetWTitle(theWindow: WindowRef; var title: Str255);
function   SetWTitle(theWindow: WindowRef; title: ConstStr255Param);
```

Displaying Windows

```
procedure  DrawGrowIcon(theWindow: WindowRef);
procedure  SelectWindow(theWindow: WindowRef);
procedure  ShowWindow(theWindow: WindowRef);
procedure  HideWindow(theWindow: WindowRef);
procedure  ShowHide(theWindow: WindowRef; showFlag: boolean);
procedure  HiliteWindow(theWindow: WindowRef; fHilite: boolean);
procedure  BringToFront(theWindow: WindowRef);
procedure  SendBehind(theWindow: WindowRef; behindWindow: WindowRef);
```

Retrieving Mouse Information

```
function   FindWindow(thePoint: Point; var theWindow: WindowRef): integer;
function   FrontWindow: WindowRef;
```

Moving Windows

```
procedure  MoveWindow(theWindow: WindowRef; hGlobal: integer; vGlobal: integer;
           front: boolean);
procedure  DragWindow(theWindow: WindowRef; startPt: Point; {CONST}var boundsRect: Rect);
function   DragGrayRgn(theRgn: RgnHandle; startPt: Point; var limitRect: Rect;
           var slopRect: Rect; axis: integer; actionProc: DragGrayRgnUPP): longint;
function   PinRect(var theRect: Rect; thePt: Point): longint;
```

Resizing Windows

```
procedure  SizeWindow(theWindow: WindowRef; w: integer; h: integer; fUpdate: boolean);
function   GrowWindow(theWindow: WindowRef; startPt: Point; var bBox: Rect): longint;
```

Zooming Windows

```
function   TrackBox(theWindow: WindowRef; thePt: Point; partCode: integer): boolean;
procedure  ZoomWindow(theWindow: WindowRef; partCode: integer; front: boolean);
```

Closing and Deallocating Windows

```
function   TrackGoAway(theWindow: WindowRef; thePt: Point): boolean;
procedure  CloseWindow(theWindow: WindowRef);
procedure  DisposeWindow(theWindow: WindowRef);
```

Maintaining the Update Region

```
procedure  BeginUpdate(theWindow: WindowRef);
procedure  EndUpdate(theWindow: WindowRef);
procedure  InvalRect(var badRect: Rect);
procedure  InvalRgn(badRgn: RgnHandle);
procedure  ValidRect(var goodRect: Rect);
procedure  ValidRgn(goodRgn: RgnHandle);
```

Setting and Retrieving Other Window Characteristics

```
procedure  SetWindowPic(theWindow: WindowRef; pic: PicHandle);
function   GetWindowPic(theWindow: WindowRef): PicHandle;
function   GetWRefCon(theWindow: WindowRef): longint;
procedure  SetWRefCon(theWindow: WindowRef; data: longint);
function   GetWVariant(theWindow: WindowRef): integer;
```

Manipulating the Desktop

```
procedure  SetDeskCPat(deskPixPat: PixPatHandle);
procedure  GetWMgrPort(var wPort: GrafPtr);
procedure  GetCWMgrPort(var wMgrCPort: CGrafPtr);
function   GetGrayRgn : RgnHandle;
```

Demonstration Program

```
{ ############################################################################################
// WindowsPascal.p
// ############################################################################################
//
// This program:
//
// •  Allows the user to open any number of zoomDocProc windows, up to the maximum
//    specified in the global variable 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 one of the system patterns 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.
//
// 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 Stop Alerts (purgeable).
//
```

```
// •  A 'SIZE' resource with the acceptSuspendResumeEvents and doesActivateOnFGSwitch
//    and is32BitCompatible flags set.
//
// ############################################################################################ }

program WindowsPascal(input, output);

{ ……… include the following Universal Interfaces }

uses

  Windows, Fonts, Menus, TextEdit, Quickdraw, Dialogs, QuickdrawText, Processes, Types,
  Memory, Events, TextUtils, ToolUtils, OSUtils, Devices, Segload;

{ ……… define the following constants }

const

 mApple = 128;
  iAbout = 1;
 mFile = 129;
 mEdit = 130;
  iNew = 1;
  iClose = 4;
  iQuit = 11;
 mWindows = 131;

 rNewWindow = 128;
 rMenubar = 128;
 rAlertBox = 128;
 rStringList = 128;
  sUntitled = 1;
  eMaxWindows = 2;
  eFailWindow = 3;
  eFailMenus = 4;
  eFailMemory = 5;

 kMaxWindows = 10;
 kMaxLong = $7FFFFFFF;

{ ……… global variables }

var

gDone : Boolean;
gInBackground : Boolean;
gPreAllocatedBlockPtr : Ptr;
gUntitledWindowNumber : longint;
gCurrentNumberOfWindows : longint;
gWindowPtrArray : array [0..kMaxWindows + 2] of WindowPtr;

menubarHdl : Handle;
menuHdl : MenuHandle;
a : integer;

{ ############################################################################ DoInitManagers }

procedure DoInitManagers;

  begin
  MaxApplZone;
  MoreMasters;
  MoreMasters;
  InitGraf(@qd.thePort);
  InitFonts;
  InitWindows;
  InitMenus;
  TEInit;
  InitDialogs(nil);

  InitCursor;
  FlushEvents(everyEvent, 0);
  end;
    {of procedure DoInitManagers}

{ ################################################################################ DoError }

```

```
116  procedure  DoError(errorType : integer);
117
118    var
119    errorMessage : string;
120    ignored : integer;
121
122    begin
123    GetIndString(errorMessage, rStringList, errorType);
124    ParamText(errorMessage, '', '', '');
125
126    if (errorType = eMaxWindows)
127      then  ignored := CautionAlert(rAlertBox, nil)
128
129      else  begin
130          ignored := StopAlert(rAlertBox, nil);
131          ExitToShell;
132          end;
133    end;
134      {of procedure DoError}
135
136  { ############################################################################ DoUpDateWindow }
137
138  procedure DoUpdateWindow(eventRec : EventRecord);
139
140    var
141    myWindowPtr : WindowPtr;
142    paintRect : Rect;
143    windowRefCon : longint;
144    fillPattern : Pattern;
145
146    begin
147    myWindowPtr := WindowPtr(eventRec.message);
148    SetPort(myWindowPtr);
149
150    paintRect := myWindowPtr^.portRect;
151    paintRect.right := paintRect.right - 15;
152    paintRect.bottom := paintRect.bottom - 15;
153
154    windowRefCon := GetWRefCon(myWindowPtr);
155
156    GetIndPattern(fillPattern, 0, windowRefCon + 9);
157
158    FillRect(paintRect, fillPattern);
159    end;
160      {of procedure DoUpdateWindow}
161
162  { ################################################################################ DoUpdate }
163
164  procedure DoUpdate(eventRec : EventRecord);
165
166    var
167    myWindowPtr : WindowPtr;
168
169    begin
170    myWindowPtr := WindowPtr(eventRec.message);
171
172    BeginUpdate(myWindowPtr);
173
174    if not (EmptyRgn(myWindowPtr^.visRgn)) then
175      begin
176      SetPort(myWindowPtr);
177      EraseRgn(myWindowPtr^.visRgn);
178      DoUpdateWindow(eventRec);
179      DrawGrowIcon(myWindowPtr);
180      end;
181
182    EndUpdate(myWindowPtr);
183    end;
184      {of procedure DoUpdate}
185
186  { ############################################################################ DoActivateWindow }
187
188  procedure DoActivateWindow(myWindowPtr : WindowPtr; becomingActive : Boolean);
189
190    var
191    windowsMenu : MenuHandle;
192    menuItem, a : integer;
```

```
193
194     begin
195     a := 1;
196
197     windowsMenu := GetMenuHandle(mWindows);
198
199     while (gWindowPtrArray[a] <> myWindowPtr) do
200       a := a + 1;
201     menuItem := a;
202
203     if (becomingActive)
204       then CheckItem(windowsMenu, menuItem, true)
205       else CheckItem(windowsMenu, menuItem, false);
206
207     DrawGrowIcon(myWindowPtr);
208     end;
209       {of procedure DoActivateWindow}
210
211   { ############################################################################ DoActivate }
212
213   procedure DoActivate(eventRec : EventRecord);
214
215     var
216     myWindowPtr : WindowPtr;
217     becomingActive : Boolean;
218
219     begin
220     myWindowPtr := WindowPtr(eventRec.message);
221
222     becomingActive := boolean(BAnd(eventRec.modifiers, activeFlag) <> 0);
223
224     DoActivateWindow(myWindowPtr, becomingActive);
225     end;
226       {of procedure DoActivate}
227
228   { ############################################################################ DoOSEvent }
229
230   procedure DoOSEvent(eventRec : EventRecord);
231
232     begin
233     case BAnd(BSR(eventRec.message, 24), $000000FF) of
234
235       suspendResumeMessage:
236         begin
237         if (gCurrentNumberOfWindows > 0) then
238           begin
239           DrawGrowIcon(FrontWindow);
240           gInBackground := boolean(BAnd(eventRec.message, resumeFlag));
241           DoActivateWindow(FrontWindow, not(gInBackground));
242           end;
243         end;
244       end;
245         {of case statement}
246     end;
247       {of procedure DoOSEvent}
248
249   { ######################################################################## SetStandardState }
250
251   procedure SetStandardState(myWindowPtr : WindowPtr);
252
253     var
254     windowRecPtr: WindowPeek;
255     winStateDataPtr : WStateDataPtr;
256     tempRect : Rect;
257
258     begin
259     tempRect := qd.screenBits.bounds;
260     windowRecPtr := WindowPeek(myWindowPtr);
261     winStateDataPtr := WStateDataPtr(Handle(windowRecPtr^.dataHandle^));
262
263     SetRect(winStateDataPtr^.stdState, tempRect.left + 40, tempRect.top + 60,
264             tempRect.right - 40, tempRect.bottom - 40);
265     end;
266       {of procedure SetStandardState}
267
268   { ############################################################################ DoNewWindow }
269
```

```
procedure DoNewWindow;

  var
  myWindowPtr : WindowPtr;
  untitledString : string;
  numberAsString : string;
  titleString : string;
  windowsMenu : MenuHandle;

  begin

  if (gCurrentNumberOfWindows = kMaxWindows) then
    begin
    DoError(eMaxWindows);
    Exit(DoNewWindow);
    end;

  myWindowPtr := GetNewWindow(rNewWindow, gPreAllocatedBlockPtr, WindowPtr(-1));
  if (myWindowPtr = nil) then
    DoError(eFailWindow);

  gPreAllocatedBlockPtr := nil;

  GetIndString(untitledString, rStringList, sUntitled);
  gUntitledWindowNumber := gUntitledWindowNumber + 1;
  NumToString(gUntitledWindowNumber, numberAsString);
  titleString := Concat(untitledString, numberAsString);
  SetWTitle(myWindowPtr, titleString);

  SetStandardState(myWindowPtr);

  ShowWindow(myWindowPtr);

  if (gUntitledWindowNumber < 10) then
    begin
    untitledString := Concat(titleString, '/');
    NumToString(gUntitledWindowNumber, numberAsString);
    titleString := Concat(untitledString, numberAsString);
    end;

  windowsMenu := GetMenu(mWindows);
  InsertMenuItem(windowsMenu, titleString, CountMItems(windowsMenu));

  SetWRefCon(myWindowPtr, gCurrentNumberOfWindows);

  gCurrentNumberOfWindows := gCurrentNumberOfWindows + 1;
  gWindowPtrArray[gCurrentNumberOfWindows] := myWindowPtr;

  if (gCurrentNumberOfWindows = 1) then
    begin
    EnableItem(GetMenu(mFile), iClose);
    EnableItem(GetMenu(mWindows), 0);
    DrawMenuBar;
    end;
  end;
    {procedure DoNewWindow}

{ ############################################################################ DoCloseWindow }

procedure DoCloseWindow;

  var
  myWindowPtr : WindowPtr;
  windowsMenu : MenuHandle;
  a : integer;

  begin
  a := 1;

  myWindowPtr := FrontWindow;
  CloseWindow(myWindowPtr);
  DisposePtr(Ptr(WindowPeek(myWindowPtr)));
  gCurrentNumberOfWindows := gCurrentNumberOfWindows - 1;

  windowsMenu := GetMenu(mWindows);
  while (gWindowPtrArray[a] <> myWindowPtr) do
    a := a + 1;
```

```
347     gWindowPtrArray[a] := nil;
348     DeleteMenuItem(windowsMenu, a);
349
350     for a := 1 to (kMaxWindows + 1) do
351       if (gWindowPtrArray[a] = nil) then
352         begin
353         gWindowPtrArray[a] := gWindowPtrArray[a + 1];
354         gWindowPtrArray[a + 1] := nil;
355         end;
356
357     if (gCurrentNumberOfWindows = 0) then
358       begin
359       DisableItem(GetMenu(mFile), iClose);
360       DisableItem(GetMenu(mWindows), 0);
361       DrawMenuBar;
362       end;
363     end;
364       {of procedure DoCloseWindow}
365
366   { ############################################################## InvalidateScrollBarArea }
367
368   procedure InvalidateScrollBarArea(myWindowPtr : WindowPtr);
369
370     var
371     tempRect : Rect;
372
373     begin
374     SetPort(myWindowPtr);
375
376     tempRect := myWindowPtr^.portRect;
377     tempRect.left := tempRect.right - 15;
378     InvalRect(tempRect);
379
380     tempRect := myWindowPtr^.portRect;
381     tempRect.top := tempRect.bottom - 15;
382     InvalRect(tempRect);
383     end;
384       {of procedure InvalidateScrollBarArea}
385
386   { ################################################################################ DoFileMenu }
387
388   procedure DoFileMenu(menuItem : integer);
389
390     begin
391     case menuItem of
392
393       iNew:
394         begin
395         DoNewWindow;
396         end;
397
398       iClose:
399         begin
400         DoCloseWindow;
401         end;
402
403       iQuit:
404         begin
405         gDone := true;
406         end;
407
408       end;
409         {of case statement}
410     end;
411       {of procedure DoFileMenu}
412
413   { ############################################################################ DoWindowsMenu }
414
415   procedure DoWindowsMenu(menuItem : integer);
416
417     var
418     myWindowPtr : WindowPtr;
419
420     begin
421     myWindowPtr := gWindowPtrArray[menuItem];
422     SelectWindow(myWindowPtr);
423     end;
```

```
424       {of procedure DoWindowsMenu}
425
426   { ############################################################################## DoMenuChoice }
427
428   procedure DoMenuChoice(menuChoice : longint);
429
430     var
431     menuID, menuItem : integer;
432     itemName : string;
433     daDriverRefNum : integer;
434
435     begin
436     menuID := HiWord(menuChoice);
437     menuItem := LoWord(menuChoice);
438
439     if (menuID = 0) then
440       Exit(DoMenuChoice);
441
442     case menuID of
443
444       mApple:
445         begin
446         if (menuItem = iAbout)
447           then  SysBeep(10)
448
449           else  begin
450               GetMenuItemText(GetMenuHandle(mApple), menuItem, itemName);
451               daDriverRefNum := OpenDeskAcc(itemName);
452               end;
453         end;
454
455       mFile:
456         begin
457         DoFileMenu(menuItem);
458         end;
459
460       mWindows:
461         begin
462         DoWindowsMenu(menuItem);
463         end;
464
465       end;
466         {of case statement}
467
468     HiliteMenu(0);
469     end;
470       {of procedure DoMenuChoice}
471
472   { ############################################################################## DoMouseDown }
473
474   procedure DoMouseDown(eventRec : EventRecord);
475
476     var
477     myWindowPtr : WindowPtr;
478     partCode : integer;
479     growRect : Rect;
480     newSize : longint;
481
482     begin
483     partCode := FindWindow(eventRec.where, myWindowPtr);
484
485     case partCode of
486
487       inMenuBar:
488         begin
489         DoMenuChoice(MenuSelect(eventRec.where));
490         end;
491
492       inSysWindow:
493         begin
494         SystemClick(eventRec, myWindowPtr);
495         end;
496
497       inContent:
498         begin
499         if (myWindowPtr <> FrontWindow) then
500           SelectWindow(myWindowPtr);
```

```
501         end;
502
503       inDrag:
504         begin
505         DragWindow(WindowRef(myWindowPtr), eventRec.where, qd.screenBits.bounds);
506         end;
507
508       inGoAway:
509         begin
510         if TrackGoAway(myWindowPtr, eventRec.where) then
511           DoCloseWindow;
512         end;
513
514       inGrow:
515         begin
516         growRect := qd.screenBits.bounds;
517         growRect.top := 80;
518         growRect.left := 160;
519         newSize := GrowWindow(myWindowPtr, eventRec.where, growRect);
520         if (newSize <> 0) then
521           begin
522           InvalidateScrollBarArea(myWindowPtr);
523           SizeWindow(myWindowPtr, LoWord(newSize), HiWord(newSize), true);
524           InvalidateScrollBarArea(myWindowPtr);
525           end;
526         end;
527
528       inZoomIn, inZoomOut:
529         begin
530         if (TrackBox(myWindowPtr, eventRec.where, partCode)) then
531           begin
532           SetPort(myWindowPtr);
533           EraseRect(myWindowPtr^.portRect);
534           ZoomWindow(myWindowPtr, partCode, false);
535           InvalRect(myWindowPtr^.portRect);
536           end;
537         end;
538
539       end;
540         {of case statement}
541     end;
542     {of procedure DoMouseDown}
543
544   { ################################################################################ DoEvents }
545
546   procedure DoEvents(eventRec : EventRecord);
547
548     var
549     charCode : char;
550
551     begin
552     case (eventRec.what) of
553       mouseDown:
554         begin
555         DoMouseDown(eventRec);
556         end;
557
558       keyDown, autoKey:
559         begin
560         charCode := chr(BAnd(eventRec.message, charCodeMask));
561         if (BAnd(eventRec.modifiers, cmdKey) <> 0) then
562           DoMenuChoice(MenuKey(charCode));
563         end;
564
565       updateEvt:
566         begin
567         DoUpdate(eventRec);
568         end;
569
570       activateEvt:
571         begin
572         DoActivate(eventRec);
573         end;
574
575       osEvt:
576         begin
577         DoOSEvent(eventRec);
```

```
578         HiliteMenu(0);
579         end;
580       end;
581         {of case statement}
582     end;
583       {of procedure DoEvents}
584   { ############################################################################## EventLoop }
585
586   procedure EventLoop;
587
588     var
589     eventRec : EventRecord;
590
591     begin
592     gDone := false;
593
594     while (not gDone) do
595       begin
596       if (WaitNextEvent(everyEvent, eventRec, kMaxLong, nil)) then
597         DoEvents(eventRec);
598
599       if (gPreAllocatedBlockPtr = nil) then
600         begin
601         gPreAllocatedBlockPtr := NewPtr(sizeof(WindowRecord));
602         if (gPreAllocatedBlockPtr = nil) then
603           DoError(eFailMemory);
604         end;
605       end;
606         {of while loop}
607     end;
608       {of procedure EventLoop}
609
610   { ################################################################## start of main program }
611
612   begin
613
614     gUntitledWindowNumber := 0;
615     gCurrentNumberOfWindows := 0;
616
617     { ……… get nonrelocatable block low in heap for first window record }
618
619     gPreAllocatedBlockPtr := NewPtr(sizeof(WindowRecord));
620     if (gPreAllocatedBlockPtr = nil) then
621       DoError(eFailMemory);
622
623     { ……… initialize managers }
624
625     DoInitManagers;
626
627     { ……… set up menu bar and menus }
628
629     menubarHdl := GetNewMBar(rMenubar);
630     if (menubarHdl = nil) then
631       DoError(eFailMenus);
632     SetMenuBar(menubarHdl);
633     DrawMenuBar;
634
635     menuHdl := GetMenuHandle(mApple);
636     if (menuHdl = nil)
637       then DoError(eFailMenus)
638       else AppendResMenu(menuHdl, 'DRVR');
639
640     { ……… initialize window pointer array }
641
642     for a := 0 to (kMaxWindows + 2) do
643       gWindowPtrArray[a] := nil;
644
645     { ……… enter eventLoop }
646
647     EventLoop;
648
649   end.
650     {of program WindowsPascal}
651
652   { ############################################################################################ }
```

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.

The constant declaration block

Lines 57-74 establish constants relating to menu IDs and resources and to resources for windows, the menu bar, an alert box, and strings to be displayed in the alert box.

kMaxWindows (Line 76) controls the maximum number of windows allowed to be open at one time. Line 77 defines kMaxLong as the maximum possible long value. This will be assigned to WaitNextEvent's sleep parameter.

The variable declaration block

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 record. gUntitledMenuNumber 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 record 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 NIL 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.

The procedure DoError

DoError displays either a Caution Alert or a Stop Alert with a specified string extracted from the 'STR#' resource identified by rStringList.

At line 123, this string is retrieved. Line 124 assigns this string to the single text replacement variable (^0) specified in the 'ALRT' resource.

If the error is simply that the maximum number of windows has been opened (Line 126), a Caution Alert is displayed (Line 127) and the function returns when the user clicks alert's OK box. If the error was such that it is pointless continuing, a Stop Alert is displayed and the program terminates when the user clicks alert's OK box (Lines 130-131).

The procedure DoUpdateWindow

DoUpdateWindow is concerned with redrawing the window's contents less the scroll bar areas. The correct graphics port is set (Line 148) before a Rect is assigned the coordinates of the graphics port portRect field as reduced to exclude the scroll bar areas Lines 150-152. Lines 154-158 then fill this area with one of the system patterns. (Of course, to speed things up, Quickdraw actually only draws that part of the Rect which equates to the visible region established by the BeginUpdate call in the DoUpdate function.)

At Line 154, the value in the window record's refCon field is retrieved. 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 for passing as the third parameter to the GetIndPattern call at Line 156, ensuring that FillRect (Line 158) has something visible and unique to draw in each window.

The procedure DoUpdate

DoUpdate attends to basic window updating. The call to BeginUpdate (Line 172) clips the visible region to the update region and then purges the update region. The call to EmptyRgn (Line 174) confirms that the visible region is not empty before the graphics port is set and the visible region is erased. The application-defined function DoUpdateWindow is then called (Line 178) to redraw the content region less the scroll bar areas. The DrawGrowIcon call completes the process by redrawing the grow icon in the size box and the scroll box area delimiting lines.

Note that the erasure of the visible region at Line 177 is necessary to account for the case of the window being resized smaller. In this case, the only area in the clipped visible region will be the scroll bar areas, which must be erased. (DrawGrowIcon draws the delimiting lines, but does not draw the background, of the scroll bar areas.)

The EndUpdate call at Line 182 restores the window's true visible region.

The procedure 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 and drawing the grow icon.

The first step in the function DoActivateWindow is to associate the received WindowPtr with its item number in the Windows menu. At Lines 199-200, 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 a variable (Line 201) used in the CheckItem calls at Lines 203-205. 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.

DrawGrowIcon is then called (Line 207) to draw the grow icon box, including the lines which delineate the scroll bar areas. Note that, if the window involved in an activate event is being deactivated, DrawGrowIcon draws the delimiting lines and an empty size box. If the window is being activated, DrawGrowIcon draws the delimiting lines and the grow icon in the size box.

The procedure DoActivate

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

The modifiers field of the event record is tested (Line 222) 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 at Line 224.

The procedure DoOSEvent

DoOSEvent handles operating system events. In this demonstration, action is taken only in the case of suspend and resume events (Line 235) and then only if at least one window is open (Line 237). The call to DrawGrowIcon at Line 239 is required regardless of whether the event is suspend or resume. If the application is about to be sent to the background, the call will draw the empty grow box in the front window. If the application is being brought to the foreground, the call will draw the grow box with the size icon in our frontmost window.

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 at Line 241 with the second parameter set to true for a resume and to false for a suspend. This will cause either Line 204 or Line 205 in the DoActivateWindow function to be

executed as appropriate. (In this demonstration, the only activation/deactivation activity is menu enabling/disabling.)

The procedure SetStandardState

The SetStandardState procedure sets the window's standard state. First the coordinates of the screen boundary are placed in a Rect (Line 259). At Line 261, the handle in the window record's dataHandle field is dereferenced to a pointer and cast to a WStateDataPtr, a pointer to a WStateData record. At Line 263, 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.

The procedure DoNewWindow

DoNewWindow opens a new window and attends to associated tasks.

Firstly, if the current number of open windows equals the maximum allowable specified by kMaxWindows, a Caution Alert is called up via the application-defined DoError function (with the string represented by eMaxWindows displayed) and an immediate return is executed when the user clicks the Alert's OK button (Lines 281-285).

At Line 287, 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 DoError procedure (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 NIL 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 (Line 291).

Lines 293-297 insert the number of the window into the title bar (for example, "Untitled 1" for the first window opened). Line 293 retrieves the string "Untitled " from the 'STR#' resource. Lines 294-295 increments the global variable which keeps track of the numbers for the title bar and converts that value to a string. Line 296 concatenates this string to the "Untitled " string. Line 297 changes the window's title and redraws the title bar.

Line 299 calls the application-defined function SetStandardState. (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.)

Line 301 makes the window visible.

Lines 303-311 add a Command key equivalent to the Windows menu item for this window. (This occurs only for the first nine opened windows.)

Line 313 assigns a value to the window record's reference constant (refCon) field. As previously stated, in this demonstration this is used to index the system's standard patterns list for a unique pattern to draw in each window's content region.

At Lines 315-316, 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.

Lines 318-323 enable the Windows menu and the Close item in the File menu when the first window is opened.

The procedure DoCloseWindow

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

At Line 339, a pointer to the frontmost window is retrieved and that window is closed by a call to CloseWindow at Line 340. CloseWindow, rather than DisposeWindow, must be used where storage for the window record was allocated manually, that is, where the second parameter in the GetNewCWindow call was not NIL. Because CloseWindow is used, the call to DisposePtr at Line 341 is necessary to dispose of the non-relocatable block occupied by the window record. With the window closed, the global variable which keeps track of the number of windows currently open is decremented (Line 342).

Lines 344-348 delete the associated menu item from the windows menu. At Lines 345-346, 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 NIL. At Line 348, the menu item is deleted.

Lines 350-355 "compact" the array, that is, move the contents of all elements above the NILed element down by one, maintaining the correlation with the Windows menu.

Lines 357-362 disable the Windows menu and the Close item in the File menu if no windows remain open as a result of this closure.

The procedure InvalidateScrollBarArea

InvalidateScrollBarArea invalidates that part of the window's content region which would be occupied by scroll bars. (Although this demonstration does not include scroll bars, this function is necessary because the windows have size boxes and the associated calls to DrawGrowIcon draw the lines which delineate the scroll bar areas in addition to the size box itself.) The function simply retrieves the coordinates of the content region 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.

The procedure DoFileMenu

DoFileMenu branches according to the File menu item choices of the user.

The procedure 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.

The procedure DoMenuChoice

DoMenuChoice branches according to the menu choices of the user.

The procedure DoMouseDown

DoMouseDown continues the processing of mouseDown events, branching according to the part code.

The inContent case (Line 497) 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 (Line 503) 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 (Line 508) 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 (Line 514) 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 Line 516, 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 (Lines 517-518).

GrowWindow (Line 519) 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 application-defined function InvalidateScrollBarArea is then called (Line 522) to create an update region comprising the old scroll bar areas.

The SizeWindow call (Line 523) 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.

A further call to InvalidateScrollBarArea (Line 524) adds the new scroll bar areas to the update region. (Note that the first InvalidateScrollBarArea call is, strictly speaking, only really required when window size has been increased, this to ensure that the "old" scroll bar areas are included in the redrawing of the update region triggered by the updateEvt arising from the size increase. Similarly, the second InvalidateScrollBarArea call is, strictly speaking, only really required when window size has been decreased. In this case, no other update region will be generated as a result of the resizing, so the InvalidateScrollBarArea call forces an updateEvt which, in this program, results in erasure of the entire window content area, redrawing of the "new" content area less the scroll bar areas, and redrawing of the "new" scroll bar delimiting lines.)

The inZoomIn and inZoomOut cases (Line 528) 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 graphics port is set to that associated with the active window and the entire content area is erased (Lines 532-533). (This erasure is not strictly necessary; it simply avoids screen flicker and some redrawing peculiarities associated with the execution of the ZoomWindow call (Line 534).)

The call to ZoomWindow redraws 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 (Line 483) and passed to ZoomWindow as its second parameter.) Finally, the InvalRect call at Line 535 triggers an updateEvt and consequential redrawing of the entire content region.

The procedure DoEvents

DoEvents branches 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 Line 578 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).

The procedure 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 kMaxLong, which is defined in the constant declaration block as the maximum possible longint value.)

At the bottom of the event loop (Lines 601-603), a new nonrelocatable block is allocated in preparation for the next window to be opened if the global variable gPreAllocatedBlockPtr contains NIL.

The main program block

Line 619 in the main function requires some explanation. When a window is created, its window record 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 records 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. Line 619 pre-allocates a nonrelocatable block which will later be used by the window record of the first window to be created. The pointer returned by the first call to GetNewWindow, which will be copied to gWindowPtrArray[1], will point to this block. gPreAllocatedBlockPtr will then be set to NIL. At the bottom of the event loop, gPreAllocatedBlockPtr will be checked. If it contains NIL, the pre-allocated block must now be occupied by a window record, in which circumstance a new block will be allocated in preparation for the next window to be opened.

> Note: It is expected that, at some point in the development of the system software, data structures such as window records will no longer be created in your application's address space. In such circumstances, the pre-allocation technique described above will be rendered improper and the correct technique will be to allow the system to itself allocate the storage for the window record by passing NULL in the second parameter of the GetNewWindow call.

If the call at Line 619 fails, Line 621 invokes an Alert box and the program terminates.

Line 625 initialises the system software managers. Note that MoreMasters must be called twice to provide sufficient master pointers for this program.

At Lines 629-638, the menus are set up. Note that error handling involving the invocation of alert boxes is introduced in this program. If an error occurs (Lines 630 and 636) the application-defined function doError will display an alert box advising of the nature of the error before terminating the program.

gWindowPtrArray[] is initialised (Lines 642-643) before the main event loop is called at Line 647.

Creating 'WIND' Resources Using ResEdit

When learning to create the major resource types in ResEdit, it is recommended that you open Macintosh C to the page containing the relevant example resource definition in Rez input format and relate what you are doing within ResEdit to that definition. Accordingly, the methodology used in the following is to "walk through" the 'WIND' resources for the Windows demonstration program, relating what you see in ResEdit to the example definitions in this chapter.

Open the chap04pascal_demo demonstration program folder and double-click on the Windows.µ.rsrc icon to start ResEdit and open Windows.µ.rsrc. The **Windows.µ.rsrc** window opens.

Double-click the WIND icon. The **WINDs from Windows.µ.rsrc** window opens. One 'WIND' resource (ID 128) appears in the list. Double-click that list entry. The **WIND ID = 128 from Windows.u.rsrc** window opens.

The following relates the example 'WIND' resource in Rez input format in this chapter to the ResEdit display and interface:

`resource 'WIND'`	This was established when the resource was created by choosing **Resource/Create New Resource**. A small dialog opened, the item **WIND** was clicked, and the dialog's **OK** button was clicked.
`(rDocWindow,`	`rDocWindow` is the 'WIND' resource ID (128). Choose **Resource/Get Resource Info**. The **Info for WIND 128 ...** window opens. Note the editable text item titled **ID:**. This is where you set the 'WIND' resource ID. ResEdit automatically assigns 128 as the 'WIND' resource ID of the first 'WIND' resource you create.
`preload, purgeable)`	While the **Info for WIND 128 ...** window is open, compare the **Attributes:** check boxes to the Resource Attributes table at Chapter 1. Note that the **Purgeable** checkbox is checked. Close the **Info for WIND 128 ...** window.
`{64,60,314,460}`	In the **WIND ID = 128 ...** window, note the **Top**, **Left**, **Bottom**, and **Right** items at the bottom left. (Note also that, in the **WIND** menu, you can change the last two items to display **Height** and **Width** if you so desire.)
`zoomDocProc`	Note that, in the row of window icons at the top of the window, the `zoomDocProc` (8) window type is highlighted. Note also that, when you choose **WIND/Set 'WIND' Characteristics...**, the **ProcID:** item in the opened dialog box shows 8. (You can set the desired Window Definition ID either here or by clicking the appropriate icon at the top of the window.) Close the dialog.
`invisible`	Back in the **WIND ID = 128 ...** window, note the check box titled **Initially Visible** at the right.

`goAway`	Note the check box titled **Close Box** at the right.
`0x0` `"untitled"`	Choose **WIND/Set 'WIND' Characteristics** again and note the items titled **refCon:** and **Window title:** Close the dialog.
`staggerParent...`	Choose **WIND/Auto Position...** and note the items chosen in the two pop-up menus.

You might also further explore the ResEdit display options by choosing **WIND/Preview at Full Size**, and the various items in the **MiniScreen** menu.

Note that, when you click on the **Color: Custom** radio button at the right of the **WIND ID = 128 ...** window, five items appear which enable you to specify colours for the various elements of the window. If you were to save the resource with this radio button set, ResEdit would automatically create a `'wctb'` (window color table) resource with the same resource ID as the associated `'WIND'` resource.

Close the **WIND ID = 128 ...** window. Close the **WINDs from Windows.µ.rsrc** window. Close the **Windows.µ.rsrc** window.