Data Structures

This section describes the Window Manager data structures: the window record, the color window record, the state data record, the window color table record, the auxiliary window record, and the window list.

A window record or color window record describes an individual window. It includes the record for the graphics port in which the window is displayed.

The state data record stores two rectangles, known as the user state and the standard state, which define the size and location of the window as specified by the user and by your application. Your application switches between the two states when the user clicks the zoom box.

A window color table defines the colors to be used for drawing the window's frame and highlighting selected text. Ordinarily, you use the default window color table, which produces windows in the colors selected by the user through the Color control panel. If your application has some unusual need to control the frame colors, you can set up your own window color tables.

The Window Manager uses auxiliary window records to associate a window with its window color table.

The Window Manager uses the window list to track all of the windows on the desktop.

The Color Window Record

The Window Manager maintains a window record or color window record for each window on the desktop.

The Window Manager supplies routines that let you access the window record as necessary. Your application seldom changes fields in the window record directly.

The `CWindowRecord` data type defines the window record for a color window. The `CWindowPeek` data type is a pointer to a color window record. The first field in the window record is in fact the record that describes the window's graphics port. The `CWindowPtr` data type is defined as a pointer to the window's graphics port.

When Color QuickDraw is not available, you can create monochrome windows using the parallel data types `WindowRecord`, `WindowPeek`, and `WindowPtr`, described in the next section, "The Window Record."

For compatibility, the `WindowPtr` and `WindowPeek` data types can point to either a color window record or a monochrome window record. You use the `WindowPtr` data type to specify a window in most Window Manager routines, and you can use it to specify a graphics port in QuickDraw routines that take the `GrafPtr` data type. Note that you can access only the fields of the window's graphics port, not the rest of the window record, through the `WindowPtr` and `CWindowPtr` data types. You use the `WindowPeek` and `CWindowPeek` data types in low-level Window Manager routines and in your own routines that access window record fields beyond the graphics port.

The routines that manipulate color windows get color information from the window color tables and the auxiliary window record described in the sections "The Window Color Table Record" on page 4-71 and "The Auxiliary Window Record" on page 4-73.

```
TYPE  CWindowPtr  = ^CGrafPtr;
      CWindowPeek = ^CWindowRecord;

TYPE  CWindowRecord =
   RECORD
      port:          CGrafPort;     {window's graphics port}
      windowKind:    Integer;       {class of the window}
      visible:       Boolean;       {visibility}
      hilited:       Boolean;       {highlighting}
      goAwayFlag:    Boolean;       {presence of close box}
      spareFlag:     Boolean;       {presence of 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 title}
      titleWidth:    Integer;       {title width in pixels}
      controlList:   ControlHandle; {handle to control list}
      nextWindow:    CWindowPeek;   {pointer to next window }
                                    { record in window list}
      windowPic:     PicHandle;     {handle to optional picture}
      refCon:        LongInt;       {storage available to your }
                                    { application}
   END;
```

Field descriptions

`port` The graphics port record that describes the graphics port in which the window is drawn.

The graphics port record, which is documented in *Inside Macintosh: Imaging*, defines the rectangle in which drawing can occur, the window's visible region, the window's clipping region, and a collection of current drawing characteristics such as fill pattern, pen location, and pen size.

`windowKind` The class of window—that is, how the window was created.

The Window Manager fills in this field when it creates the window record. It places a negative value in `windowKind` when the window

was created by a desk accessory. (The value is the reference ID of the desk accessory.) This field can also contain one of two constants:

```
CONST
   dialogKind  = 2;     {dialog or alert window}
   userKind    = 8;     {window created by an }
                        { application}
```

The value `dialogKind` identifies all dialog or alert box windows, whether created by the system software or, indirectly through the Dialog Manager, by your application. The Dialog Manager uses this field to help it track dialog and alert box windows.

The value `userKind` represents a window created directly by your application.

`visible` A Boolean value indicating whether or not the window is visible. If the window is visible, the Window Manager sets this field to `TRUE`; if not, `FALSE`. Visibility means only whether or not the window is to be displayed, not necessarily whether you can see it on the screen. (For example, a window that is completely covered by other windows can still be visible, even if the user cannot see it on the screen.)

`hilited` A Boolean value indicating whether the window is highlighted—that is, drawn with stripes in the title bar. Only the active window is ordinarily highlighted. When the window is highlighted, the `hilited` field contains `TRUE`; when not, `FALSE`.

`goAwayFlag` A Boolean value indicating whether the window has a close box.

The Window Manager fills in this field when it creates the window according to the information in the `'WIND'` resource or the parameters passed to the function that creates the window.

If the value of `goAwayFlag` is `TRUE`, and if the window type supports a close box, the Window Manager draws a close box when the window is highlighted.

`spareFlag` A Boolean value indicating whether the window type supports zooming. The Window Manager sets this field to `TRUE` if the window's type is one that includes a zoom box (`zoomDocProc`, `zoomNoGrow`, or even `modalDBoxProc + zoomDocProc`).

`strucRgn` A handle to the structure region, which is defined in global coordinates. The structure region is the entire screen area covered by the window—that is, both the window contents and the window frame.

`contRgn` A handle to the content region, which is defined in global coordinates. The content region is the part of the window that contains the document, dialog, or other data; the window controls; and the size box.

`updateRgn` A handle to the update region, which is defined in global coordinates. The update region is the portion of the window that must be redrawn. It is maintained jointly by the Window Manager and your application. The update region excludes parts of the window that are covered by other windows.

`windowDefProc` A handle to the definition function that controls the window.

There's no need for your application to access this field directly.

In Macintosh models that use only 24-bit addressing, this field contains both a handle to the window's definition function and the window's variation code. If you need to know the variation code, regardless of the addressing mode, call the `GetWVariant` function.

`dataHandle` Usually a handle to a data area used by the window definition function.

For zoomable windows, `dataHandle` contains a handle to the `WStateData` record, which contains the user state and standard state rectangles. The `WStateData` record is described in "The Window State Data Record" beginning on page 4-70.

A window definition function that needs only 4 bytes of data can use the `dataHandle` field directly, instead of storing a handle to the data. The window definition function that handles rounded-corner windows, for example, stores the diameters of curvature in the `dataHandle` field.

`titleHandle` A handle to the string that defines the title of the window.

`titleWidth` The width, in pixels, of the window's title.

`controlList` A handle to the window's control list, which is used by the Control Manager. (See the chapter "Control Manager" in this book for a description of control lists.)

`nextWindow` A pointer to the next window in the window list, that is, the window behind this window on the desktop. In the window record for the last window on the desktop, the `nextWindow` field is set to `NIL`.

`windowPic` A handle to a QuickDraw picture of the window's contents. The Window Manager initially sets the `windowPic` field to `NIL`. If you're using the window to display a stable image, you can use the `SetWindowPic` procedure to place a handle to the picture in this field. When the window's contents need updating, the Window Manager then redraws the contents itself instead of generating an update event.

`refCon` The window's reference value field, which is simply storage space available to your application for any purpose. The sample code in this chapter uses the `refCon` field to associate a window with the data it displays by storing a window type constant in the `refCon` field of alert and dialog window records and a handle to a document record in the `refCon` field of a document window record.

Note

The close box, drag region, zoom box, and size box are not included in the window record because they don't necessarily have the formal data structure for regions as defined in QuickDraw. The window definition function determines where these regions are. ◆

The Window Record

If Color QuickDraw is not available, you create windows with a parallel data structure, the window record. The only difference between a color window record and a window record is that a color window record points to a color graphics port, which allows full use of Macintosh computers with color capability, and a window record points to a monochrome graphics port

The data types that describe window records, `WindowRecord`, `WindowPtr`, and `WindowPeek`, are parallel to the data types that describe color window records, and the fields in the monochrome window record are identical to the fields in the color window record. For a complete description, see "The Color Window Record" beginning on page 4-65.

```
TYPE  WindowPtr   = ^GrafPtr;
      WindowPeek  = ^WindowRecord;

TYPE  WindowRecord =                {all fields have same use }
   RECORD                           { as in color window record}
      port:          GrafPort;      {window's graphics port}
      windowKind:    Integer;       {class of the window}
      visible:       Boolean;       {visibility}
      hilited:       Boolean;       {highlighting}
      goAwayFlag:    Boolean;       {presence of close box}
      spareFlag:     Boolean;       {presence of 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 title}
      titleWidth:    Integer;       {title width in pixels}
      controlList:   ControlHandle; {handle to control list}
      nextWindow:    WindowPeek;    {pointer to next window }
                                    { record in window list}
      windowPic:     PicHandle;     {handle to optional picture}
      refCon:        LongInt;       {storage available to your }
                                    { application}
   END;
```

The Window State Data Record

The zoom box allows the user to alternate quickly between two window positions and sizes: the user state and the standard state. The Window Manager stores the user state and your application stores the standard state in the window state data record, whose handle appears in the `dataHandle` field of the window record.

The `WStateData` record data type defines the window state data record.

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

      WStateData =
      RECORD
         userState:  Rect; {size and location established by user}
         stdState:   Rect; {size and location established by app}
      END;
```

Field descriptions

`userState` A rectangle that describes the window size and location established by the user.

The Window Manager initializes the user state to the size and location of the window when it is first displayed, and then updates the `userState` field whenever the user resizes a window. Although the user state specifies both the size and location of the window, the Window Manager updates the state data record only when the user resizes a window—not when the user merely moves a window.

`stdState` The rectangle describing the window size and location that your application considers the most convenient, considering the function of the document, the screen space available, and the position of the window in its user state. If your application does not define a standard state, the Window Manager automatically sets the standard state to the entire gray region on the main screen, minus a three-pixel border on all sides. The user cannot change a window's standard state.

Your application typically calculates and sets the standard state each time the user zooms to the standard state. In a word-processing application, for example, a standard state window might show a full page, if possible, or a page of full width and as much length as fits on the screen. If the user changes the page size through Page Setup, the application might adjust the standard state to reflect the new page size. (See *Macintosh Human Interface Guidelines* for a detailed description of how your application determines where to open and zoom windows.)

The `ZoomWindow` procedure changes the size of a window according to the values in the window state data record. The procedure changes the window to the user state when the user zooms "in" and to the standard state when the user zooms "out." For a detailed

description of zooming windows, see "Zooming a Window" beginning on page 4-53. For descriptions of the routines you call when zooming windows, see "Zooming Windows" beginning on page 4-101.

The Window Color Table Record

The user controls the colors used for the window frame and text highlighting through the Color control panel. Ordinarily, your application doesn't override the user's color choices, which are stored in a default window color table. If you have some extraordinary need to control window colors, you can do so by defining window color tables for your application's windows.

The Window Manager maintains window color information tables in a data structure of type `WinCTab`.

You can define your own window color table and apply it to an existing window through the `SetWinColor` procedure.

To establish the window color table for a window when you create it, you provide a window color table (`'wctb'`) resource with the same resource ID as the `'WIND'` resource that defines the window.

The `WCTabPtr` data type is a pointer to a window color table record, and the `WTabHandle` is a handle to a window color table record.

```
TYPE  WCTabPtr = ^WinCTab;
      WCTabHandle = ^WCTabPtr;
```

The `WinCTab` data type defines a window color table record.

```
TYPE  WinCTab =
      RECORD
         wCSeed:     LongInt;    {reserved}
         wCReserved: Integer;    {reserved}
         ctSize:     Integer;    {number of entries in table -1}
         ctTable:    ARRAY[0..4] OF ColorSpec;
                                 {array of color specification }
                                 { records}
      END;
```

Field descriptions

`wCSeed`	Reserved.
`wCReserved`	Reserved.
`ctSize`	The number of entries in the table, minus 1. If you're building a color table for use with the standard window definition function, the maximum value of this field is 12. Custom window definition functions can use color tables of any size.

`ctTable` An array of `colorSpec` records.

In a window color table, each `colorSpec` record specifies a window part in the first word and an RGB value in the other three words:

```
TYPE  ColorSpec =
      RECORD
         value:   Integer;    {part identifier}
         rgb:     RGBColor;   {RGB value}
      END;
```

The `value` field of a `colorSpec` record specifies a constant that defines which part of the window the color controls. For the window color table used by the standard window definition function, you can specify these values with these meanings:

```
CONST
wContentColor     = 0;  {content region background}
wFrameColor       = 1;  {window outline}
wTextColor        = 2;  {window title and button }
                        { text}
wHiliteColor      = 3;  {reserved}
wTitleBarColor    = 4;  {reserved}
wHiliteColorLight = 5;  {lightest stripes in }
                        { title bar and lightest }
                        { dimmed text}
wHiliteColorDark  = 6;  {darkest stripes in }
                        { title bar and }
                        { darkest dimmed }
                        { text}
wTitleBarLight    = 7;  {lightest parts of }
                        { title bar background}
wTitleBarDark     = 8;  {darkest parts of }
                        { title bar background}
wDialogLight      = 9;  {lightest element }
                        { of dialog box frame}
wDialogDark       = 10; {darkest element of }
                        { dialog box frame}
wTingeLight       = 11; {lightest window tinging}
wTingeDark        = 12; {darkest window tinging}
```

Note

The part codes in System 5 and System 6 are significantly different from the part codes described here, which apply only to System 7. ◆

The window parts can appear in any order in the table.

The `rgb` field of a `ColorSpec` record contains three words of data that specify the red, green, and blue values of the color to be used. The `RGBColor` data type is defined in *Inside Macintosh: Imaging.*

When your application creates a window, the Window Manager first looks for a resource of type `'wctb'` with the same resource ID as the `'WIND'` resource used for the window. If it finds one, it creates a window color table for the window from the information in that resource, and then displays the window in those colors. If it doesn't find a window color table resource with the same resource ID as your window resource, the Window Manager uses the default system window color table, read into the heap during application startup.

After creating a window, you can change the entries in a window's window color table with the `SetWinColor` procedure, described on page 4-114.

See "The Window Color Table Resource" on page 4-127 for a description of the window color table resource.

The Auxiliary Window Record

The auxiliary window record specifies the color table used by a window and contains reference information used by the Dialog Manager and the Window Manager.

The Window Manager creates and maintains the information in an auxiliary window record; your application seldom, if ever, needs to access an auxiliary window record.

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

      AuxWinRec =
      RECORD
         awNext:        AuxWinHandle;  {handle to next record}
         awOwner:       WindowPtr;     {pointer to window }
                                       { associated with this }
                                       { record}
         awCTable:      CTabHandle;    {handle to color table}
         dialogCItem:   Handle;        {storage used by }
                                       { Dialog Manager}
         awFlags:       LongInt;       {reserved}
         awReserved:    CTabHandle;    {reserved}
         awRefCon:      LongInt;       {reference constant, }
                                       { for application's use}
      END;
```

Field descriptions

`awNext` A handle to the next record in the auxiliary window list, used by the Window Manager to maintain the auxiliary window list as a linked list. If a window is using the default auxiliary window record, this value is `NIL`.

`awOwner` A pointer to the window that uses this record. The `awOwner` field of the default auxiliary window record is set to `NIL`.

`awCTable`	A handle to the window's color table. Unless you specify otherwise, this is a handle to the system window color table.
`dialogCItem`	Private storage for use by the Dialog Manager.
`awFlags`	Reserved.
`awReserved`	Reserved.
`awRefCon`	The reference constant, typically used by an application to associate the auxiliary window record with a document record.

Except in unusual circumstances, your application doesn't need to manipulate window color tables or the auxiliary window record.

For compatibility with other applications in the shared environment, your application should not manipulate system color tables directly but should go through the Palette Manager, documented in *Inside Macintosh: Imaging*. If your application provides its own window and control definition functions, these functions should apply the user's desktop color choices the same way the standard window and control definition functions do.

The Window List

The Window Manager maintains information about the windows on the desktop in a private structure called the *window list*. The window list contains pointers to all windows on the desktop, both visible and invisible, and contains other information that the Window Manager uses to maintain the desktop.

Your application should not directly access the information in a window list. The structure of the window list is private to the Window Manager.

The global variable `WindowList` contains a pointer to the first window in the window list.

Window Manager Routines

This section describes the complete set of routines for creating, displaying, and managing windows.

Initializing the Window Manager

Before using any other other Window Manager routines, you must initialize the Window Manager by calling the `InitWindows` procedure.

As part of initialization, `InitWindows` creates the **Window Manager port,** a graphics port that occupies all of the main screen. The Window Manager port is named `WMgrCPort` on Macintosh computers equipped with Color QuickDraw and `WMgrPort` on computers with only QuickDraw.

Ordinarily, your application does not need to know about the Window Manager port. If necessary, however, you can retrieve a pointer to it by calling the procedure `GetWMgrPort` or `GetCWMgrPort`. Your application should not draw directly into the Window Manager port, except through custom window definition functions.

The Window Manager draws your application's windows into the Window Manager port. The port rectangle of the Window Manager port is the bounding rectangle of the main screen (`screenBits.bounds`). To accommodate systems with multiple monitors, QuickDraw recognizes a port rectangle of `screenBits.bounds` as a special case and allows drawing on all parts of the desktop.

InitWindows

The procedure `InitWindows` initializes the Window Manager for your application. Before calling `InitWindows`, you must initialize QuickDraw and the Font Manager by calling the `InitGraf` and `InitFonts` procedures, documented in *Inside Macintosh: Imaging* and *Inside Macintosh: Text*.

```
PROCEDURE InitWindows;
```

DESCRIPTION

The `InitWindows` procedure initializes the Window Manager.

ASSEMBLY-LANGUAGE INFORMATION

When the desktop needs to be redrawn any time after initialization, the Window Manager checks the global variable `DeskHook`, which can be used as a pointer to an application-defined routine for drawing the desktop. This variable is ordinarily set to 0, but not until after system startup. If you're displaying windows in code that is to be executed during startup, set `DeskHook` to 0. Note that the use of the Window Manager's global variables is not guaranteed to be compatible in system software versions later than System 6.

Creating Windows

You can create windows in two ways:

- from a window resource (a resource of type `'WIND'`), with the `GetNewCWindow` and `GetNewWindow` functions
- from a collection of window characteristics passed as parameters to the `NewCWindow` and `NewWindow` functions

Creating windows from resources allows you to localize your application for different languages and to change the characteristics of your windows during application development by changing only the window resources.

All four functions, `GetNewCWindow`, `GetNewWindow`, `NewCWindow`, and `NewWindow`, can allocate space in your application's heap for the new window's window record. For more control over memory use, you can allocate the space yourself and pass a pointer when creating a window. In either case, the Window Manager fills in the data structure and returns a pointer to it.

GetNewCWindow

Use the `GetNewCWindow` function to create a color window with the properties defined in the `'WIND'` resource with a specified resource ID.

```
FUNCTION GetNewCWindow (windowID: Integer; wStorage: Ptr;
                        behind: WindowPtr): WindowPtr;
```

`windowID` The resource ID of the `'WIND'` resource that defines the properties of the window.

`wStorage` A pointer to memory space for the window record.

If you specify a value of `NIL` for `wStorage`, the `GetNewCWindow` function allocates the window record as a nonrelocatable object in the heap. You can reduce the chances of heap fragmentation by allocating the memory your application needs for window records early in your initialization code. Whenever you need to create a window, you can allocate memory from your own block and pass a pointer to it in the `wStorage` parameter.

`behind` A pointer to the window that appears immediately in front of the new window on the desktop.

To place a new window in front of all other windows on the desktop, specify a value of `Pointer(-1)`. When you place a window in front of all others, `GetNewCWindow` removes the highlighting from the previously active window, highlights the newly created window, and generates the appropriate activate events. Note that if you create an invisible window in front of all others on the desktop, the user sees no active window until you make the new window visible (or make another window active).

To place a new window behind all other windows, specify a value of `NIL`.

DESCRIPTION

The `GetNewCWindow` function creates a new color window from the specified window resource and returns a pointer to the newly created window record. You can use the returned window pointer to refer to this window in most Window Manager routines. If `GetNewCWindow` is unable to read the window or window definition function from the resource file, it returns `NIL`.

The `GetNewCWindow` function looks for a `'wctb'` resource with the same resource ID as that of the `'WIND'` resource. If it finds one, it uses the window color information in the `'wctb'` resource for coloring the window frame and highlighting selected text.

If the window's definition function (specified in the window resource) is not already in memory, `GetNewCWindow` reads it into memory and stores a handle to it in the window record. It allocates space in the application heap for the structure and content regions of the window and asks the window definition function to calculate those regions.

To create the window, `GetNewCWindow` retrieves the window characteristics from the window resource and then calls the `NewCWindow` function, passing the characteristics as parameters.

The `GetNewCWindow` function creates a window in a color graphics port. Before calling `GetNewCWindow`, verify that Color QuickDraw is available. Your application typically sets up its own global variables reflecting the system setup during initialization by calling the `Gestalt` function. See *Inside Macintosh: Overview* for more information about establishing the local configuration.

SPECIAL CONSIDERATIONS

Note that the `GetNewCWindow` function returns a value of type `WindowPtr`, not `CWindowPtr`.

If you let the Window Manager create the window record in your application's heap, call `DisposeWindow` to dispose of the window's window record. If you allocated the memory for the window record yourself and passed a pointer to the storage to `GetNewCWindow`, use the procedure `CloseWindow` to close the window and the procedure `DisposePtr`, documented in *Inside Macintosh: Memory*, to dispose of the window record.

SEE ALSO

See Listing 4-3 on page 4-28 for an example that calls `GetNewCWindow` to create a new window from a window resource.

For more information about window characteristics and the window resource, see the description of `NewCWindow` beginning on page 4-79 and the description of the `'WIND'` resource in the section "The Window Resource" beginning on page 4-124.

For the procedures for closing a window and removing the structures from memory, see the descriptions of the `DisposeWindow` procedure on page 4-105, the `CloseWindow` procedure on page 4-104, and the `DisposePtr` procedure in *Inside Macintosh: Memory*. See Listing 4-17 on page 4-61 for an example of closing a document window.

GetNewWindow

Use the `GetNewWindow` function to create a new window from a window resource when Color QuickDraw is not available. The `GetNewWindow` function takes the same parameters as `GetNewCWindow` and returns a value of type `WindowPtr`. The only difference is that it creates a monochrome graphics port, not a color graphics port. The window record and graphics port record that describe monochrome and color graphics ports are the same size and can be used interchangeably in most Window Manager routines.

```
FUNCTION GetNewWindow (windowID: Integer; wStorage: Ptr;
                       behind: WindowPtr): WindowPtr;
```

`windowID` The resource ID of the `'WIND'` resource that defines the properties of the window.

`wStorage` A pointer to memory space for the window record.

If you specify a value of `NIL` for `wStorage`, the `GetNewWindow` function allocates the window record as a nonrelocatable object in the heap. You can reduce the chances of heap fragmentation by allocating the memory your application needs for window records early in your initialization code. Whenever you need to create a window, you can allocate memory from your own block and pass a pointer to it in the `wStorage` parameter.

`behind` A pointer to the window that appears immediately in front of the new window on the desktop.

To place a new window in front of all other windows on the desktop, specify a value of `Pointer(-1)`. When you place a window in front of all others, `GetNewWindow` removes the highlighting from the previously active window, highlights the newly created window, and generates the appropriate activate events. Note that if you create an invisible window in front of all others on the desktop, the user sees no active window until you make the new window visible (or make another window active).

To place a new window behind all other windows, specify a value of `NIL`.

DESCRIPTION

Like `GetNewCWindow`, `GetNewWindow` creates a new window from a window resource, but it creates a monochrome window. The `GetNewWindow` function creates a new window from the specified window resource and returns a pointer to the newly created window record. You can use the returned window pointer to refer to this window in most Window Manager routines. If `GetNewWindow` is unable to read the window or window definition function from the resource file, it returns `NIL`.

If the window's definition function (specified in the window resource) is not already in memory, `GetNewWindow` reads it into memory and stores a handle to it in the window record. It allocates space in the application heap for the structure and content regions of the window and asks the window definition function to calculate those regions.

To create the window, `GetNewWindow` retrieves the window characteristics from the window resource and then calls the function `NewWindow`, passing the characteristics as parameters.

SPECIAL CONSIDERATIONS

If you let the Window Manager create the window record in your application's heap, call `DisposeWindow` to dispose of the window's window record. If you allocated the memory for the window record yourself and passed a pointer to `GetNewWindow`, use the procedure `CloseWindow` to close the window and the procedure `DisposePtr`, documented in *Inside Macintosh: Memory*, to dispose of the window record.

SEE ALSO

For more information about window characteristics and the window resource, see the description of `NewWindow` beginning on page 4-82 and the description of the `'WIND'` resource in the section "The Window Resource" beginning on page 4-124.

For the procedures for closing a window and removing the structures from memory, see the descriptions of the `DisposeWindow` procedure on page 4-105, the `CloseWindow` procedure on page 4-104, and the `DisposePtr` procedure in *Inside Macintosh: Memory*.

NewCWindow

You can use the `NewCWindow` function to create a window with a specified list of characteristics.

```
FUNCTION NewCWindow (wStorage: Ptr; boundsRect: Rect;
                     title: Str255; visible: Boolean;
                     procID: Integer; behind: WindowPtr;
                     goAwayFlag: Boolean;
                     refCon: LongInt): WindowPtr;
```

`wStorage` A pointer to the window record. If you specify `NIL` as the value of `wStorage`, `NewCWindow` allocates the window record as a nonrelocatable object in the application heap. You can reduce the chances of heap fragmentation by allocating memory from a block of memory reserved for this purpose by your application and passing a pointer to it in the `wStorage` parameter.

`boundsRect` A rectangle, in global coordinates, specifying the window's initial size and location. This parameter becomes the port rectangle of the window's graphics port. For the standard window types, the `boundsRect` field defines the content region of the window. The `NewCWindow` function places the origin of the local coordinate system at the upper-left corner of the port rectangle.

Note

The `NewCWindow` function actually calls the QuickDraw procedure `OpenCPort` to create the graphics port. The bitmap, pen pattern, and other characteristics of the window's graphics port are the same as the default values set by `OpenCPort`, except for the character font, which is set to the application font instead of the system font. ◆

`title` A string that specifies the window's title.

If the title is too long to fit in the title bar, the title is truncated. If the window has a close box, characters are truncated at the end of the title; if there's no close box, the title is centered and truncated at both ends.

To suppress the title in a window with a title bar, pass an empty string, not `NIL`, in the `title` parameter.

`visible` A Boolean value indicating visibility status: `TRUE` means that the Window Manager displays the window; `FALSE` means it does not.

If the value of the `visible` parameter is `TRUE`, the Window Manager draws a new window as soon as the window exists. The Window Manager first calls the window definition function to draw the window frame. If the value of the `goAwayFlag` parameter is also `TRUE` and the window is frontmost (that is, if the value of the `behind` parameter is `Pointer(-1)`), the Window Manager instructs the window definition function to draw a close box in the window frame. After drawing the frame, the Window Manager generates an update event to trigger your application's drawing of the content region.

When you create a window, you typically specify `FALSE` as the value of the `visible` parameter. When you're ready to display the window, you call the `ShowWindow` procedure, described on page 4-88.

`procID` The window's definition ID, which specifies both the window definition function and the variation code within that definition function.

The Window Manager supports nine standard window types, which are handled by two window definition functions. You can create windows of the standard types by specifying one of the window definition ID constants:

```
CONST
   documentProc      = 0;  {standard document }
                           { window, no zoom box}
   dBoxProc          = 1;  {alert box or modal }
                           { dialog box}
   plainDBox         = 2;  {plain box}
   altDBoxProc       = 3;  {plain box with shadow}
   noGrowDocProc     = 4;  {movable window, }
                           { no size box or zoom box}
   movableDBoxProc   = 5;  {movable modal dialog box}
   zoomDocProc       = 8;  {standard document window}
   zoomNoGrow        = 12; {zoomable, nonresizable }
                           { window}
   rDocProc          = 16; {rounded-corner window}
```

For a description of the nine standard window types, see "Types of Windows" beginning on page 4-8.

You can control the diameter of curvature of rounded-corner windows by adding an integer to the `rDocProc` constant, as described in "The Window Resource" beginning on page 4-124.

`behind` A pointer to the window that appears immediately in front of the new window on the desktop.

To place a new window in front of all other windows on the desktop, specify a value of `Pointer(-1)`. When you place a new window in front of all others, `NewCWindow` removes highlighting from the previously active window, highlights the newly created window, and generates activate events that trigger your application's updating of both windows. Note that if you create an invisible window in front of all others on the desktop, the user sees no active window until you make the new window visible (or make another window active).

To place a new window behind all other windows, specify a value of `NIL`.

`goAwayFlag` A Boolean value that determines whether the window has a close box. If the value of `goAwayFlag` is `TRUE` and the window type supports a close box, the Window Manager draws a close box in the title bar and recognizes mouse clicks in the close region; if the value of `goAwayFlag` is `FALSE` or the window type does not support a close box, it does not.

`refCon` The window's reference constant, set and used only by your application. (See "Managing Multiple Windows" beginning on page 4-23 for some suggested ways to use the `refCon` parameter.)

DESCRIPTION

The `NewCWindow` function creates a window as specified by its parameters, adds it to the window list, and returns a pointer to the newly created window record. You can use the returned window pointer to refer to this window in most Window Manager routines. If `NewCWindow` is unable to read the window definition function from the resource file, it returns `NIL`.

The `NewCWindow` function looks for a `'wctb'` resource with the same resource ID as the `'WIND'` resource. If it finds one, it uses the window color information in the `'wctb'` resource for coloring the window frame and highlighting.

If the window's definition function is not already in memory, `NewCWindow` reads it into memory and stores a handle to it in the window record. It allocates space for the structure and content regions of the window and asks the window definition function to calculate those regions.

Storing the characteristics of your windows as resources, especially window titles and window items, makes your application easier to localize.

The `NewCWindow` function creates a window in a color graphics port. Creating color windows whenever possible ensures that your windows appear on color monitors with whatever color options the user has selected. Before calling `GetNewCWindow`, verify that Color QuickDraw is available. Your application typically sets up its own set of global

variables reflecting the system setup during initialization by calling the `Gestalt` function. See the chapter *Inside Macintosh: Overview* for more information about establishing the local configuration.

Note that the function `NewCWindow` returns a value of type `WindowPtr`, not `CWindowPtr`.

SPECIAL CONSIDERATIONS

If you let the Window Manager create the window record in your application's heap, call the `DisposeWindow` procedure to close the window and dispose of its window record. If you allocated the memory for the window record yourself and passed a pointer to `NewCWindow`, use the `CloseWindow` procedure to close the window and the `DisposePtr` procedure, documented in *Inside Macintosh: Memory*, to dispose of the window record.

SEE ALSO

For the procedures for closing a window and removing the structures from memory, see the descriptions of the `DisposeWindow` procedure on page 4-105, the `CloseWindow` procedure on page 4-104, and the `DisposePtr` procedure in *Inside Macintosh: Memory*.

NewWindow

Use the `NewWindow` function to create a new window with the characteristics specified by a list of parameters when Color QuickDraw is not available. The `NewWindow` function takes the same parameters as `NewCWindow` and, like `NewCWindow`, returns a `WindowPtr` as its function result. The only difference is that `NewWindow` creates a window in a monochrome graphics port, not a color graphics port. The window record and graphics port record that describe monochrome and color graphics ports are the same size and can be used interchangeably in most Window Manager routines.

```
FUNCTION NewWindow (wStorage: Ptr; boundsRect: Rect;
                    title: Str255; visible: Boolean;
                    theProc: Integer; behind: WindowPtr;
                    goAwayFlag: Boolean;
                    refCon: LongInt): WindowPtr;
```

`wStorage` A pointer to the window record. If you specify `NIL` as the value of `wStorage`, `NewWindow` allocates the window record as a nonrelocatable object in the heap. You can reduce the chances of heap fragmentation by allocating the storage from a block of memory reserved for this purpose by your application and passing a pointer to it in the `wStorage` parameter.

`boundsRect` A rectangle, in global coordinates, specifying the window's initial size and location. This parameter becomes the port rectangle of the window's graphics port. For the standard window types, `boundsRect` defines the content region of the window. The `NewWindow` function places the origin of the local coordinate system at the upper-left corner of the port rectangle.

Note

The `NewWindow` function actually calls the QuickDraw procedure `OpenPort` to create the graphics port. The bitmap, pen pattern, and other characteristics of the window's graphics port are the same as the default values set by `OpenPort`, except for the character font, which is set to the application font instead of the system font. The coordinates of the graphics port's port boundaries and visible region are changed along with its port rectangle. ◆

`title` A string that specifies the window's title.

If the title is too long to fit in the title bar, the title is truncated. If the window has a close box, characters at the end of the title are truncated; if there's no close box, the title is centered and truncated at both ends.

To suppress the title in a window with a title bar, pass an empty string, not `NIL`.

`visible` A Boolean value indicating visibility status: `TRUE` means that the Window Manager displays the window; `FALSE` means it does not.

If the value of the `visible` parameter is `TRUE`, the Window Manager draws a new window as soon as the window exists. The Window Manager first calls the window definition function to draw the window frame. If the value of the `goAwayFlag` parameter (described below) is also `TRUE` and the window is frontmost (that is, if the value of the `behind` parameter is `Pointer(-1)`), the Window Manager instructs the window definition function to draw a close box in the window frame. After drawing the frame, the Window Manager generates an update event to trigger your application's drawing of the content region.

When you create a window, you typically specify `FALSE` as the value of the `visible` parameter. When you're ready to display the window, you call the `ShowWindow` procedure, described on page 4-88.

`theProc` The window's definition ID, which specifies both the window definition function and the variation code for that definition function.

The Window Manager supports nine standard window types, which are handled by two window definition functions. You can create windows of the standard types by specifying one of the type constants:

```
CONST
   documentProc      = 0;  {standard document }
                           { window, no zoom box}
   dBoxProc          = 1;  {alert box or modal }
                           { dialog box}
   plainDBox         = 2;  {plain box}
```

```
altDBoxProc       = 3;  {plain box with shadow}
noGrowDocProc     = 4;  {movable window, }
                        { no size box or zoom box}
movableDBoxProc   = 5;  {movable modal dialog box}
zoomDocProc       = 8;  {standard document window}
zoomNoGrow        = 12; {zoomable, nonresizable }
                        { window}
rDocProc          = 16; {rounded-corner window}
```

You can control the diameter of curvature of rounded-corner windows by adding an integer to the `rDocProc` constant, as described in "The Window Resource" beginning on page 4-124.

`behind` A pointer to the window that appears immediately in front of the new window on the desktop.

To place a new window in front of all other windows on the desktop, specify a value of `Pointer(-1)`. When you place a new window in front of all others, `NewWindow` removes highlighting from the previously active window, highlights the newly created window, and generates activate events that trigger your application's updating of both windows. Note that if you create an invisible window in front of all others on the desktop, the user sees no active window until you make the new window visible (or make another window active).

To place a new window behind all other windows, specify a value of `NIL`.

`goAwayFlag` A Boolean value that determines whether or not the window has a close box. If the value of `goAwayFlag` is `TRUE` and the window type supports a close box, the Window Manager draws a close box in the title bar and recognizes mouse clicks in the close region; if the value of `goAwayFlag` is `FALSE` or the window type does not support a close box, it does not.

`refCon` The window's reference constant, set and used only by your application. (See "Managing Multiple Windows" beginning on page 4-23 for some suggested ways to use the `refCon` parameter.)

DESCRIPTION

The `NewWindow` function creates a window as specified by its parameters, adds it to the window list, and returns a pointer to the newly created window record. You can use the returned window pointer to refer to this window in most Window Manager routines. If `NewWindow` is unable to read the window definition function from the resource file, it returns `NIL`.

If the window's definition function is not already in memory, `NewWindow` reads it into memory and stores a handle to it in the window record. It allocates space for the structure and content regions of the window and asks the window definition function to calculate those regions.

Storing the characteristics of your windows as resources, especially window titles and window items, makes your application easier to localize.

SPECIAL CONSIDERATIONS

If you let the Window Manager create the window record in your application's heap, call the `DisposeWindow` procedure to close the window and dispose of its window record. If you allocated the memory for the window record yourself and passed a pointer to `NewCWindow`, use the `CloseWindow` procedure to close the window and the `DisposePtr` procedure, documented in *Inside Macintosh: Memory*, to dispose of the window record.

SEE ALSO

For the procedures for closing a window and removing the structures from memory, see the descriptions of the `DisposeWindow` procedure on page 4-105, the `CloseWindow` procedure on page 4-104, and the `DisposePtr` procedure in *Inside Macintosh: Memory*.

Naming Windows

This section describes the procedures that set and retrieve a window's title.

SetWTitle

Use the `SetWTitle` procedure to change a window's title.

```
PROCEDURE SetWTitle (theWindow: WindowPtr; title: Str255);
```

`theWindow` A pointer to the window's window record.
`title` The new window title.

DESCRIPTION

The `SetWTitle` procedure changes a window's title to the specified string, both in the window record and on the screen, and redraws the window's frame as necessary.

When the user opens a previously saved document, you typically create a new (invisible) window with the title "untitled" and then call `SetWTitle` to give the window the document's name before displaying it. You also call `SetWTitle` when the user saves a document under a new name.

To suppress the title in a window with a title bar, pass an empty string, not `NIL`.

Always use `SetWTitle` instead of directly changing the title in a window's window record.

GetWTitle

Use the `GetWTitle` procedure to retrieve a window's title.

```
PROCEDURE GetWTitle (theWindow: WindowPtr; VAR title: Str255);
```

`theWindow` A pointer to the window record.
`title` The window title.

DESCRIPTION

The `GetWTitle` procedure returns the title of the window in the `title` parameter.

Your application seldom needs to determine a window's title. It might need to do so, however, when presenting user dialog boxes during operations that can affect multiple files. A spell-checking command, for example, might display a dialog box that lets the user select from all currently open documents.

When you need to retrieve a window's title, you should always use `GetWTitle` instead of reading the title from a window's window record.

Displaying Windows

This section describes the Window Manager routines that change a window's display and position in the window list but not its size or location on the desktop. Note that the Window Manager automatically draws all visible windows on the screen.

Your application typically uses only a few of the routines described in this section: `DrawGrowIcon`, `SelectWindow`, `ShowWindow`, and, occasionally, `HideWindow`.

DrawGrowIcon

Use the `DrawGrowIcon` procedure to draw a window's size box.

```
PROCEDURE DrawGrowIcon (theWindow: WindowPtr);
```

`theWindow` A pointer to the window record.

DESCRIPTION

The `DrawGrowIcon` procedure draws a window's size box or, if the window can't be sized, whatever other image is appropriate. You call `DrawGrowIcon` when drawing the content region of a window that contains a size box.

The exact appearance and location of the image depend on the window type and the window's active or inactive state. The `DrawGrowIcon` procedure automatically checks the window's type and state and draws the appropriate image.

In an active document window, `DrawGrowIcon` draws the grow image in the size box in the lower-right corner of the window's graphics port rectangle, along with the lines delimiting the size box and scroll bar areas. To draw the size box but not the scroll bar outline, set the `clipRgn` field in the window's graphics port to be a 15-by-15 pixel rectangle in the lower-right corner of the window.

The `DrawGrowIcon` procedure doesn't erase the scroll bar areas. If you use `DrawGrowIcon` to draw the size box and scroll bar outline, therefore, you should erase those areas yourself when the window size changes, even if the window doesn't contain scroll bars.

In an inactive document window, `DrawGrowIcon` draws the lines delimiting the size box and scroll bar areas and erases the size box.

SEE ALSO

See Listing 4-8 on page 4-39 for an example that draws a window's content region, including the size box. See Listing 4-11 on page 4-51 for an example that calls `DrawGrowIcon` to remove the size-box icon when a window becomes inactive.

SelectWindow

Use the `SelectWindow` procedure to make a window active. The `SelectWindow` procedure changes the active status of a window but does not affect its visibility.

```
PROCEDURE SelectWindow (theWindow: WindowPtr);
```

`theWindow` A pointer to the window's window record.

DESCRIPTION

The `SelectWindow` procedure removes highlighting from the previously active window, brings the specified window to the front, highlights it, and generates the activate events to deactivate the previously active window and activate the specified window. If the specified window is already active, `SelectWindow` has no effect.

Even if the specified window is invisible, `SelectWindow` brings the window to the front, activates the window, and deactivates the previously active window. Note that in this case, no active window is visible on the screen. If you do select an invisible window, be sure to call `ShowWindow` immediately to make the window visible (and accessible to the user).

Call `SelectWindow` when the user presses the mouse button while the cursor is in the content region of an inactive window.

SEE ALSO

See Listing 4-9 on page 4-44 for an example that calls `SelectWindow` to change the active window when the user presses the mouse button while the cursor is in an inactive window.

See Listing 4-18 on page 4-64 for an example that uses `SelectWindow` and `ShowWindow` together to restore a window's active, visible status after it has been made invisible with `HideWindow`.

ShowWindow

Use the `ShowWindow` procedure to make an invisible window visible.

```
PROCEDURE ShowWindow (theWindow: WindowPtr);
```

`theWindow` A pointer to the window record of the window.

DESCRIPTION

The `ShowWindow` procedure makes an invisible window visible. If the specified window is already visible, `ShowWindow` has no effect. Your application typically creates a new window in an invisible state, performs any necessary setup of the content region, and then calls `ShowWindow` to make the window visible.

When you display a previously invisible window by calling `ShowWindow`, the Window Manager draws the window frame and then generates an update event to trigger your application's drawing of the content region.

If the newly visible window is the frontmost window, `ShowWindow` highlights it if it's not already highlighted and generates an activate event to make it active. The `ShowWindow` procedure does not activate a window that is not frontmost on the desktop.

Note
Because `ShowWindow` does not change the front-to-back ordering of windows, it is not the inverse of `HideWindow`. If you make the frontmost window invisible with `HideWindow`, and `HideWindow` has activated another window, you must call both `ShowWindow` and `SelectWindow` to bring the original window back to the front. ◆

SEE ALSO

See Listing 4-16 on page 4-60 for an example that temporarily hides a dialog box window when the user closes it. See Listing 4-18 on page 4-64 for the example that calls `ShowWindow` to display the window again later.

HideWindow

Use the `HideWindow` procedure to make a window invisible.

```
PROCEDURE HideWindow (theWindow: WindowPtr);
```

`theWindow` A pointer to the window's window record.

DESCRIPTION

The `HideWindow` procedure make a visible window invisible. If you hide the frontmost window, `HideWindow` removes the highlighting, brings the window behind it to the front, highlights the new frontmost window, and generates the appropriate activate events.

To reverse the actions of `HideWindow`, you must call both `ShowWindow`, to make the window visible, and `SelectWindow`, to select it.

SEE ALSO

See Listing 4-16 on page 4-60 for an example that calls `HideWindow` to temporarily hide a dialog box window when the user closes it. See Listing 4-18 on page 4-64 for the companion example that redisplays the window later.

ShowHide

Use the `ShowHide` procedure to set a window's visibility status.

```
PROCEDURE ShowHide (theWindow: WindowPtr; showFlag: Boolean);
```

`theWindow` A pointer to the window's window record.

`showFlag` A Boolean value that determines visibility status: `TRUE` makes a window visible; `FALSE` makes it invisible.

DESCRIPTION

The `ShowHide` procedure sets a window's visibility to the status specified by the `showFlag` parameter. If the value of `showFlag` is `TRUE`, `ShowHide` makes the window visible if it's not already visible and has no effect if it's already visible. If the value of `showFlag` is `FALSE`, `ShowHide` makes the window invisible if it's not already invisible and has no effect if it's already invisible.

The `ShowHide` procedure never changes the highlighting or front-to-back ordering of windows and generates no activate events.

▲ **WARNING**
Use this procedure carefully and only in special circumstances where you need more control than that provided by `HideWindow` and `ShowWindow`. Do not, for example, use `ShowHide` to hide the active window without making another window active. ▲

HiliteWindow

Use the `HiliteWindow` procedure to set a window's highlighting status.

```
PROCEDURE HiliteWindow (theWindow: WindowPtr; fHilite: Boolean);
```

`theWindow` A pointer to the window's window record.

`fHilite` A Boolean value that determines the highlighting status: `TRUE` highlights a window; `FALSE` removes highlighting.

DESCRIPTION

The `HiliteWindow` procedure sets a window's highlighting status to the specified state. If the value of the `fHilite` parameter is `TRUE`, `HiliteWindow` highlights the specified window; if the specified window is already highlighted, the procedure has no effect. If the value of `fHilite` is `FALSE`, `HiliteWindow` removes highlighting from the specified window; if the window is not already highlighted, the procedure has no effect.

Your application doesn't normally need to call `HiliteWindow`. To make a window active, you can call `SelectWindow`, which handles highlighting for you.

BringToFront

Use the `BringToFront` procedure to bring a window to the front.

```
PROCEDURE BringToFront (theWindow: WindowPtr);
```

`theWindow` A pointer to the window's window record.

DESCRIPTION

The `BringToFront` procedure puts the specified window at the beginning of the window list and redraws the window in front of all others on the screen. It does not change the window's highlighting or make it active.

Your application does not ordinarily call `BringToFront`. The user interface guidelines specify that the frontmost window should be the active window. To bring a window to the front and make it active, call the `SelectWindow` procedure.

SendBehind

Use the `SendBehind` procedure to move one window behind another.

```
PROCEDURE SendBehind (theWindow, behindWindow: WindowPtr);
```

`theWindow` A pointer to the window to be moved.

`behindWindow`
A pointer to the window that is to be in front of the moved window.

DESCRIPTION

The `SendBehind` procedure moves the window pointed to by the parameter `theWindow` behind the window pointed to by the parameter `behindWindow`. If the move exposes previously obscured windows or parts of windows, `SendBehind` redraws the frames as necessary and generates the appropriate update events to have any newly exposed content areas redrawn.

If the value of `behindWindow` is `NIL`, `SendBehind` sends the window to be moved behind all other windows on the desktop. If the window to be moved is the active window, `SendBehind` removes its highlighting, highlights the newly exposed frontmost window, and generates the appropriate activate events.

Note
Do not use `SendBehind` to deactivate a window after you've made a new window active with the `SelectWindow` procedure. The `SelectWindow` procedure automatically deactivates the previously active window. ◆

Retrieving Window Information

This section describes

- the `FindWindow` function, which maps the cursor location of a mouse-down event to parts of the screen or regions of a window
- the `FrontWindow` function, which tells your application which window is active

FindWindow

When your application receives a mouse-down event, call the `FindWindow` function to map the location of the cursor to a part of the screen or a region of a window.

```
FUNCTION FindWindow (thePoint: Point;
                     VAR theWindow: WindowPtr): Integer;
```

`thePoint` The point, in global coordinates, where the mouse-down event occurred. Your application retrieves this information from the `where` field of the event record.

`theWindow` A parameter in which `FindWindow` returns a pointer to the window in which the mouse-down event occurred, if it occurred in a window. If it didn't occur in a window, `FindWindow` sets `theWindow` to `NIL`.

DESCRIPTION

The `FindWindow` function returns an integer that specifies where the cursor was when the user pressed the mouse button. You typically call `FindWindow` whenever you receive a mouse-down event. The `FindWindow` function helps you dispatch the event by reporting whether the cursor was in the menu bar or in a window when the mouse button was pressed and, if it was in a window, which window and which region of the window. If the mouse-down event occurred in a window, `FindWindow` places a pointer to the window in the parameter `theWindow`.

The `FindWindow` function returns an integer that specifies one of nine regions:

```
CONST inDesk      = 0;  {none of the following}
      inMenuBar   = 1;  {in menu bar}
      inSysWindow = 2;  {in desk accessory window}
      inContent   = 3;  {anywhere in content region except size }
                        { box if window is active, }
                        { anywhere including size box if window }
                        { is inactive}
      inDrag      = 4;  {in drag (title bar) region}
      inGrow      = 5;  {in size box (active window only)}
      inGoAway    = 6;  {in close box}
      inZoomIn    = 7;  {in zoom box (window in standard state)}
      inZoomOut   = 8;  {in zoom box (window in user state)}
```

The `FindWindow` function returns `inDesk` if the cursor is not in the menu bar, a desk accessory window, or any window that belongs to your application. The `FindWindow` function might return this value if, for example, the user presses the mouse button while the cursor is on the window frame but not in the title bar, close box, or zoom box. When `FindWindow` returns `inDesk`, your application doesn't need to do anything. In System 7, when the user presses the mouse button while the cursor is on the desktop or in a window that belongs to another application, the Event Manager sends your application a suspend event and switches to the Finder or another application.

The `FindWindow` function returns `inMenuBar` when the user presses the mouse button with the cursor in the menu bar. Your application typically adjusts its menus and then calls the Menu Manager's function `MenuSelect` to let the user choose menu items.

The FindWindow function returns `inSysWindow` when the user presses the mouse button while the cursor is in a window belonging to a desk accessory that was launched in your application's partition. This situation seldom arises in System 7. When the user

clicks in a window belonging to a desk accessory launched independently, the Event Manager sends your application a suspend event and switches to the desk accessory.

If `FindWindow` does return `inSysWindow`, your application calls the `SystemClick` procedure, documented in the chapter "Event Manager" in this book. The `SystemClick` procedure routes the event to the desk accessory. If the user presses the mouse button with the cursor in the content region of an inactive desk accessory window, `SystemClick` makes the window active by sending your application and the desk accessory the appropriate activate events.

The `FindWindow` function returns `inContent` when the user presses the mouse button with the cursor in the content area (excluding the size box in an active window) of one of your application's windows. Your application then calls its routine for handling clicks in the content region.

The `FindWindow` function returns `inDrag` when the user presses the mouse button with the cursor in the drag region of a window (that is, the title bar, excluding the close box and zoom box). Your application then calls the Window Manager's `DragWindow` procedure to let the user drag the window to a new location.

The `FindWindow` function returns `inGrow` when the user presses the mouse button with the cursor in an active window's size box. Your application then calls its own routine for resizing a window.

The `FindWindow` function returns `inGoAway` when the user presses the mouse button with the cursor in an active window's close box. Your application calls the `TrackGoAway` function to track mouse activity while the button is down and then calls its own routine for closing a window if the user releases the button while the cursor is in the close box.

The `FindWindow` function returns `inZoomIn` or `inZoomOut` when the user presses the mouse button with the cursor in an active window's zoom box. Your application calls the `TrackBox` function to track mouse activity while the button is down and then calls its own routine for zooming a window if the user releases the button while the cursor is in the zoom box.

SEE ALSO

See Listing 4-9 on page 4-44 for an example that calls `FindWindow` to determine the location of the cursor and then dispatches the mouse-down event depending on the results.

FrontWindow

Use the `FrontWindow` function to find out which window is active.

```
FUNCTION FrontWindow: WindowPtr;
```

DESCRIPTION

The `FrontWindow` function returns a pointer to the first visible window in the window list (that is, the active window). If there are no visible windows, `FrontWindow` returns `NIL`.

SEE ALSO

See Listing 4-9 on page 4-44 for an example that calls `FrontWindow` to determine whether an event occurred in the active window.

See Listing 4-12 on page 4-55 for an example that calls `FrontWindow` to determine whether to display a window in front of other windows after changing its size.

See Listing 4-16 on page 4-60 and Listing 4-17 on page 4-61 for examples that call `FrontWindow` to determine which window is affected by a user command directed at the active window.

Moving Windows

This section describes the procedures that move windows on the desktop.

To move a window, your application ordinarily needs to call only the `DragWindow` procedure, which itself calls the `DragGrayRgn` function, and the `MoveWindow` procedure. The `DragGrayRgn` function drags a dotted outline of the window on the screen, following the motion of the cursor, as long as the user holds down the mouse button. The `DragGrayRgn` function itself calls the `PinRect` function to contain the point where the cursor was when the mouse button was first pressed inside the available desktop area. When the user releases the mouse button, `DragWindow` calls `MoveWindow`, which moves the window to a new location.

DragWindow

When the user drags a window by its title bar, use the `DragWindow` procedure to move the window on the screen.

```
PROCEDURE DragWindow (theWindow: WindowPtr;
                      startPt: Point; boundsRect: Rect);
```

`theWindow` A pointer to the window record of the window to be dragged.

`startPt` The location, in global coordinates, of the cursor at the time the user pressed the mouse button. Your application retrieves this point from the `where` field of the event record.

`boundsRect` A rectangle, in global coordinates, that limits the region to which a window can be dragged. If the mouse button is released when the cursor is outside the limits of `boundsRect`, `DragWindow` returns without moving the window (or, if it was inactive, without making it the active window).

Because the user cannot ordinarily move the cursor off the desktop, you can safely set `boundsRect` to the largest available rectangle (the bounding box of the desktop region pointed to by the global variable `GrayRgn`) when you're using `DragWindow` to track mouse movements. Don't set the bounding rectangle to the size of the immediate screen (`screenBits.bounds`), because the user wouldn't be able to move the window to a different screen on a system equipped with multiple monitors.

DESCRIPTION

The `DragWindow` procedure moves a dotted outline of the specified window around the screen, following the movement of the cursor until the user releases the mouse button. When the button is released, `DragWindow` calls `MoveWindow` to move the window to its new location. If the specified window isn't the active window (and the Command key wasn't down when the mouse button was pressed), `DragWindow` makes it the active window by setting the `front` parameter to `TRUE` when calling `MoveWindow`. If the Command key was down when the mouse button was pressed, `DragWindow` moves the window without making it active.

SEE ALSO

The `DragWindow` procedure calls both `MoveWindow` and `DragGrayRgn`, which are described in this section.

See Listing 4-9 on page 4-44 for an example that calls `DragWindow` when the user presses the mouse button while the cursor is in the drag region.

MoveWindow

Use the `MoveWindow` procedure to move a window on the desktop.

```
PROCEDURE MoveWindow (theWindow: WindowPtr;
                      hGlobal, vGlobal: Integer;
                      front: Boolean);
```

`theWindow` A pointer to the window record of the window being moved.

`hGlobal` The new location, in global coordinates, of the left edge of the window's port rectangle.

`vGlobal` The new location, in global coordinates, of the top edge of the window's port rectangle.

`front` A Boolean value specifying whether the window is to become the frontmost, active window. If the value of the front parameter is `FALSE`, `MoveWindow` does not change its plane or status. If the value of the front parameter is `TRUE` and the window isn't active, `MoveWindow` makes it active by calling the `SelectWindow` procedure.

DESCRIPTION

The `MoveWindow` procedure moves the specified window to the location specified by the `hGlobal` and `vGlobal` parameters, without changing the window's size. The upper-left corner of the window's port rectangle is placed at the point (`vGlobal`,`hGlobal`). The local coordinates of the upper-left corner are unaffected.

Your application doesn't normally call `MoveWindow`. When the user drags a window by dragging its title bar, you can call `DragWindow`, which in turn calls `MoveWindow` when the user releases the mouse button.

DragGrayRgn

The `DragWindow` function calls the `DragGrayRgn` function to move an outline of a window around the screen as the user drags a window.

```
FUNCTION DragGrayRgn (theRgn: RgnHandle; startPt: Point;
                      limitRect, slopRect: Rect; axis: Integer;
                      actionProc: ProcPtr): LongInt;
```

`theRgn` A handle to the region to be dragged.

`startPt` The location, in the local coordinates of the current graphics port, of the cursor when the mouse button was pressed.

`limitRect` A rectangle, in the local coordinates of the current graphics port, that limits where the region can be dragged. This parameter works in conjunction with the `slopRect` parameter, as illustrated in Figure 4-23 on page 4-98.

`slopRect` A rectangle, in the local coordinates of the current graphics port, that gives the user some leeway in moving the mouse without violating the limits of the `limitRect` parameter, as illustrated in Figure 4-23 on page 4-98. The `slopRect` rectangle should be larger than the `limitRect` rectangle.

`axis` A constant that constrains the region's motion. The `axis` parameter can have one of these values:

```
CONST noConstraint   = 0;  {no constraints}
      hAxisOnly      = 1;  {move on horizontal axis }
                           { only}
      vAxisOnly      = 2;  {move on vertical axis }
                           { only}
```

If an axis constraint is in effect, the outline follows the cursor's movements along only the specified axis, ignoring motion along the other axis. With or without an axis constraint, the outline appears only when the mouse is inside the `slopRect` rectangle.

`actionProc` A pointer to a procedure that defines an action to be performed repeatedly as long as the user holds down the mouse button. The procedure can have no parameters. If the value of `actionProc` is `NIL`, `DragGrayRgn` simply retains control until the mouse button is released.

DESCRIPTION

The `DragGrayRgn` function moves a gray outline of a region on the screen, following the movements of the cursor, until the mouse button is released. It returns the difference between the point where the mouse button was pressed and the **offset point**—that is, the point in the region whose horizontal and vertical offsets from the upper-left corner of the region's enclosing rectangle are the same as the offsets of the starting point when the user pressed the mouse button. The `DragGrayRgn` function stores the vertical difference between the starting point and the offset point in the high-order word of the return value and the horizontal difference in the low-order word.

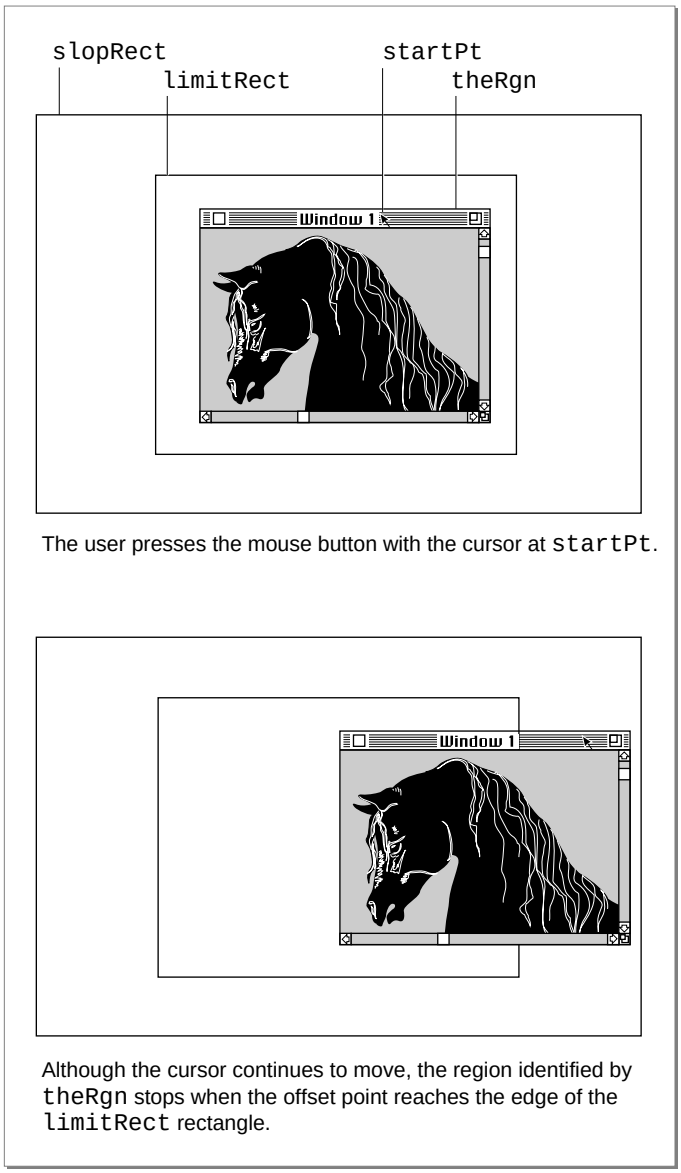
The `DragGrayRgn` function limits the movement of the region according to the constraints set by the `limitRect` and `slopRect` parameters:

- As long as the cursor is inside the `limitRect` rectangle, the region's outline follows it normally. If the mouse button is released while the cursor is within this rectangle, the return value reflects the simple distance that the cursor moved in each dimension.
- When the cursor moves outside the `limitRect` rectangle, the offset point stops at the edge of the `limitRect` rectangle. If the mouse button is released while the cursor is outside the `limitRect` rectangle but inside the `slopRect` rectangle, the return value reflects only the difference between the starting point and the offset point, regardless of how far outside of the `limitRect` rectangle the cursor may have moved. (Note that part of the region can fall outside the `limitRect` rectangle, but not the offset point.)
- When the cursor moves outside the `slopRect` rectangle, the region's outline disappears from the screen. The `DragGrayRgn` function continues to track the cursor, however, and if the cursor moves back into the `slopRect` rectangle, the outline reappears. If the mouse button is released while the cursor is outside the `slopRect` rectangle, both words of the return value are set to $8000. In this case, the Window Manager does not move the window from its original location.

Figure 4-23 on page 4-98 illustrates how the region stops moving when the offset point reaches the edge of the `limitRect` rectangle. The cursor continues to move, but the region does not.

If the mouse button is released while the cursor is anywhere inside the `slopRect` rectangle, the Window Manager redraws the window in its new location, which is calculated from the value returned by `DragGrayRgn`.

Figure 4-23 Limiting rectangle used by `DragGrayRgn`



ASSEMBLY-LANGUAGE INFORMATION

You can set the global variable `DragHook` to point to an optional procedure, defined by your application, which will be called by `DragGrayRgn` as long as the mouse button is held down. (If there's an `actionProc` procedure, it is called first.) If you want `DragGrayRgn` to draw the region's outline in a pattern other than gray, you can store the pattern in the global variable `DragPattern` and then invoke the macro `_DragTheRgn`. Note that the use of the Window Manager's global variables is not guaranteed to be compatible with system software versions later than System 6.

PinRect

The `DragGrayRgn` function uses the `PinRect` function to contain a point within a specified rectangle.

```
FUNCTION PinRect (theRect: Rect; thePt: Point): LongInt;
```

`theRect` The rectangle in which the point is to be contained.

`thePt` The point to be contained.

DESCRIPTION

The `PinRect` function returns a point within the specified rectangle that is as close as possible to the specified point. (The high-order word of the returned long integer is the vertical coordinate; the low-order word is the horizontal coordinate.)

If the specified point is within the rectangle, `PinRect` returns the point itself. If not, then

- if the horizontal position is to the left of the rectangle, `PinRect` returns the left edge as the horizontal coordinate
- if the horizontal position is to the right of the rectangle, `PinRect` returns the right edge minus 1 as the horizontal coordinate
- if the vertical position is above the rectangle, `PinRect` returns the top edge as the vertical coordinate
- if the vertical position is below the rectangle, `PinRect` returns the bottom edge minus 1 as the vertical coordinate

Note
The 1 is subtracted when the point is below or to the right of the rectangle so that a pixel drawn at that point lies within the rectangle. If the point is exactly on the bottom or the right edge of the rectangle, however, 1 should be subtracted but isn't. ◆

Resizing Windows

This section describes the procedures you can use to track the cursor while the user resizes a window and to draw the window in a new size.

GrowWindow

Use the `GrowWindow` function to allow the user to change the size of a window. The `GrowWindow` function displays an outline (grow image) of the window as the user moves the cursor to make the window larger or smaller; it handles all user interaction

until the user releases the mouse button. After calling `GrowWindow`, you call the `SizeWindow` procedure to change the size of the window.

```
FUNCTION GrowWindow (theWindow: WindowPtr;
                     startPt: Point; sizeRect: Rect): LongInt;
```

`theWindow` A pointer to the window record of the window to drag.

`startPt` The location of the cursor at the time the mouse button was first pressed, in global coordinates. Your application retrieves this point from the `where` field of the event record.

`sizeRect` The limits on the vertical and horizontal measurements of the port rectangle, in pixels.

Although the `sizeRect` parameter is in the form of the `Rect` data type, the four numbers in the structure represent lengths, not screen coordinates. The `top`, `left`, `bottom`, and `right` fields of the `sizeRect` parameter specify the minimum vertical measurement (`top`), the minimum horizontal measurement (`left`), the maximum vertical measurement (`bottom`), and the maximum horizontal measurement (`right`).

The minimum measurements must be large enough to allow a manageable rectangle; 64 pixels on a side is typical. Because the user cannot ordinarily move the cursor off the screen, you can safely set the upper bounds to the largest possible length (65,535 pixels) when you're using `GrowWindow` to follow cursor movements.

DESCRIPTION

The `GrowWindow` function moves a dotted-line image of the window's right and lower edges around the screen, following the movements of the cursor until the mouse button is released. It returns the new dimensions, in pixels, of the resulting window: the height in the high-order word of the returned long-integer value and the width in the low-order word. You can use the functions `HiWord` and `LoWord` to retrieve only the high-order and low-order words, respectively.

A return value of 0 means that the new size is the same as the size of the current port rectangle.

ASSEMBLY-LANGUAGE INFORMATION

You can set the global variable `DragHook` to point to an optional procedure, defined by your application, which will be called by `GrowWindow` as long as the mouse button is held down. (If there's an `actionProc` procedure, the `actionProc` procedure is called first.) Note that the use of the Window Manager's global variables is not guaranteed to be compatible with system software versions later than System 6.

SEE ALSO

See Listing 4-13 on page 4-58 for an example that calls `GrowWindow` when the user presses the mouse button while the cursor is in the size box.

SizeWindow

Use the `SizeWindow` procedure to set the size of a window.

```
PROCEDURE SizeWindow (theWindow: WindowPtr; w, h: Integer;
                      fUpdate: Boolean);
```

`theWindow`	A pointer to the window record of the window to be sized.
w	The new window width, in pixels.
`h`	The new window height, in pixels.
`fUpdate`	A Boolean value that specifies whether any newly created area of the content region is to be accumulated into the update region (`TRUE`) or not (`FALSE`). You ordinarily pass a value of `TRUE` to ensure that the area is updated. If you pass `FALSE`, you're responsible for maintaining the update region yourself. For more information on adding rectangles to and removing rectangles from the update region, see the description of `InvalRect` on page 4-107 and `ValidRect` on page 4-108.

DESCRIPTION

The `SizeWindow` procedure changes the size of the window's graphics port rectangle to the dimensions specified by the `w` and `h` parameters, or does nothing if the values of `w` and `h` are 0. The Window Manager redraws the window in the new size, recentering the title and truncating it if necessary. Your application calls `SizeWindow` immediately after calling `GrowWindow`, to adjust the window to any changes made by the user through the size box.

SEE ALSO

See Listing 4-13 on page 4-58 for an example that calls `SizeWindow` to resize a window based on the return value of `GrowWindow`.

Zooming Windows

This section describes the procedures you can use to track mouse activity in the zoom box and to zoom windows.

TrackBox

Use the `TrackBox` function to track the cursor when the user presses the mouse button while the cursor is in the zoom box.

```
FUNCTION TrackBox (theWindow: WindowPtr; thePt: Point;
                   partCode: Integer): Boolean;
```

`theWindow`	A pointer to the window record of the window in which the mouse button was pressed.
`thePt`	The location of the cursor when the mouse button was pressed. Your application receives this point from the `where` field in the event record.
`partCode`	The part code (either `inZoomIn` or `inZoomOut`) returned by the `FindWindow` function.

DESCRIPTION

The `TrackBox` function tracks the cursor when the user presses the mouse button while the cursor is in the zoom box, retaining control until the mouse button is released. While the button is down, `TrackBox` highlights the zoom box while the cursor is in the zoom region, as illustrated in Figure 4-20 on page 4-47.

When the mouse button is released, `TrackBox` removes the highlighting from the zoom box and returns `TRUE` if the cursor is within the zoom region and `FALSE` if it is not.

Your application calls the `TrackBox` function when it receives a result code of either `inZoomIn` or `inZoomOut` from the `FindWindow` function. If `TrackBox` returns `TRUE`, your application calculates the standard state, if necessary, and calls the `ZoomWindow` procedure to zoom the window. If `TrackBox` returns `FALSE`, your application does nothing.

ASSEMBLY-LANGUAGE INFORMATION

You can set the global variable `DragHook` to point to an optional procedure, defined by your application, which will be called by `TrackBox` as long as the mouse button is held down. (If there's an `actionProc` procedure, the `actionProc` procedure is called first.) Note that the use of the Window Manager's global variables is not guaranteed to be compatible with system software versions later than System 6.

SEE ALSO

See Listing 4-12 on page 4-55 for an example that calls `TrackBox` to track cursor activity when the user presses the mouse button while the cursor is in the zoom box.

ZoomWindow

Use the `ZoomWindow` procedure to zoom the window when the user has pressed and released the mouse button with the cursor in the zoom box.

```
PROCEDURE ZoomWindow (theWindow: WindowPtr;
                      partCode: Integer; front: Boolean);
```

`theWindow`	A pointer to the window record of the window to be zoomed.
`partCode`	The result (either `inZoomIn` or `inZoomOut`) returned by the `FindWindow` function.

`front` A Boolean value that determines whether the window is to be brought to the front. If the value of `front` is `TRUE`, the window necessarily becomes the frontmost, active window. If the value of `front` is `FALSE`, the window's position in the window list does not change. Note that if a window was active before it was zoomed, it remains active even if the value of `front` is `FALSE`.

DESCRIPTION

The `ZoomWindow` procedure zooms a window in or out, depending on the value of the `partCode` parameter. Your application calls `ZoomWindow`, passing it the part code returned by `FindWindow`, when it receives a result of `TRUE` from `TrackBox`. The `ZoomWindow` procedure then changes the window's port rectangle to either the user state (if the part code is `inZoomIn`) or the standard state (if the part code is `inZoomOut`), as stored in the window state data record, described in the section "Zooming a Window" beginning on page 4-53.

If the part code is `inZoomOut`, your application ordinarily calculates and sets the standard state before calling `ZoomWindow`.

For best results, call the QuickDraw procedure `EraseRect`, passing the window's graphics port as the port rectangle, before calling `ZoomWindow`.

SEE ALSO

See Listing 4-12 on page 4-55 for an example that calculates and sets the standard state and then calls `ZoomWindow` to zoom a window.

Closing and Deallocating Windows

This section describes the procedures that track user activity in the close box and that close and dispose of windows.

When you no longer need a window, call the `CloseWindow` procedure if you allocated the memory for the window record or the `DisposeWindow` procedure if you did not.

TrackGoAway

Use the `TrackGoAway` function to track the cursor when the user presses the mouse button while the cursor is in the close box.

```
FUNCTION TrackGoAway (theWindow: WindowPtr;
                      thePt: Point): Boolean;
```

`theWindow` A pointer to the window record of the window in which the mouse-down event occurred.

`thePt` The location of the cursor at the time the mouse button was pressed. Your application receives this point from the `where` field of the event record.

DESCRIPTION

The `TrackGoAway` function tracks cursor activity when the user presses the mouse button while the cursor is in the close box, retaining control until the user releases the mouse button. While the button is down, `TrackGoAway` highlights the close box as long as the cursor is in the close region, as illustrated in Figure 4-19 on page 4-46.

When the mouse button is released, `TrackGoAway` removes the highlighting from the close box and returns `TRUE` if the cursor is within the close region and `FALSE` if it is not.

Your application calls the `TrackGoAway` function when it receives a result code of `inGoAway` from the `FindWindow` function. If `TrackGoAway` returns `TRUE`, your application calls its own procedure for closing a window, which can call either the `CloseWindow` procedure or the `DisposeWindow` procedure to remove the window from the screen. (Before removing a document window, your application ordinarily checks whether the document has changed since the associated file was last saved. See the chapter "Introduction to File Management" in *Inside Macintosh: Files* for a general discusion of handling files.) If `TrackGoAway` returns `FALSE`, your application does nothing.

ASSEMBLY-LANGUAGE INFORMATION

You can set the global variable `DragHook` to point to an optional procedure, defined by your application, which will be called by `TrackGoAway` as long as the mouse button is held down. (If there's an `actionProc` procedure, the `actionProc` procedure is called first.) Note that the use of the Window Manager's global variables is not guaranteed to be compatible with system software versions later than System 6.

SEE ALSO

See Listing 4-9 on page 4-44 for an example that calls `TrackGoAway` to track cursor activity when the user presses the mouse button while the cursor is in the close box.

CloseWindow

Use the `CloseWindow` procedure to remove a window if you allocated memory yourself for the window's window record.

```
PROCEDURE CloseWindow (theWindow: WindowPtr);
```

`theWindow` A pointer to the window record of the window to be closed.

DESCRIPTION

The `CloseWindow` procedure removes the specified window from the screen and deletes it from the window list. It releases the memory occupied by all data structures associated with the window except the window record itself.

If you allocated memory for the window record and passed a pointer to it as one of the parameters to the functions that create windows, call `CloseWindow` when you're done with the window. You must then call the Memory Manager procedure `DisposePtr` to release the memory occupied by the window record.

▲ **WARNING**
If your application allocated any other memory for use with a window, you must release it before calling `CloseWindow`. The Window Manager releases only the data structures it created.

Also, `CloseWindow` assumes that any picture pointed to by the window record field `windowPic` is data, not a resource, and it calls the QuickDraw procedure `KillPicture` to delete it. If your application uses a picture stored as a resource, you must release the memory it occupies with the `ReleaseResource` procedure and set the `windowPic` field to `NIL` before closing the window. ▲

Any pending update events for the window are discarded. If the window being removed is the frontmost window, the window behind it, if any, becomes the active window.

SEE ALSO

See Listing 4-17 on page 4-61 for an example that calls `CloseWindow` to remove a window from the screen.

See Listing 4-3 on page 4-28 for an example that calls `CloseWindow` to clean up memory when an attempt to create a new window fails.

DisposeWindow

Use the `DisposeWindow` procedure to remove a window if you let the Window Manager allocate memory for the window record.

```
PROCEDURE DisposeWindow (theWindow: WindowPtr);
```

`theWindow` A pointer to the window record of the window to be closed.

DESCRIPTION

The `DisposeWindow` procedure removes a window from the screen, deletes it from the window list, and releases the memory occupied by all structures associated with the window, including the window record. (`DisposeWindow` calls `CloseWindow` and then releases the memory occupied by the window record.)

▲ **WARNING**

If your application allocated any other memory for use with a window, you must release it before calling `DisposeWindow`. The Window Manager releases only the data structures it created.

The `DisposeWindow` procedure assumes that any picture pointed to by the window record field `windowPic` is data, not a resource, and it calls the QuickDraw procedure `KillPicture` to delete it. If your application uses a picture stored as a resource, you must release the memory it occupies with the `ReleaseResource` procedure and set the `windowPic` field to `NIL` before closing the window. ▲

Any pending update events for the window are discarded. If the window being removed is the frontmost window, the window behind it, if any, becomes the active window.

Maintaining the Update Region

This section describes the routines you use to update your windows and to maintain window update regions.

BeginUpdate

Use the `BeginUpdate` procedure to start updating a window when you receive an update event for that window.

```
PROCEDURE BeginUpdate (theWindow: WindowPtr);
```

`theWindow` A pointer to the window's window record. Your application gets this information from the `message` field in the update event record.

DESCRIPTION

The `BeginUpdate` procedure limits the visible region of the window's graphics port to the intersection of the visible region and the update region; it then sets the window's update region to an empty region. After calling `BeginUpdate`, your application redraws either the entire content region or only the visible region. In either case, only the parts of the window that require updating are actually redrawn on the screen.

Every call to `BeginUpdate` must be matched with a subsequent call to `EndUpdate` after your application redraws the content region.

Note

In Pascal, `BeginUpdate` and `EndUpdate` can't be nested. That is, you must call `EndUpdate` before the next call to `BeginUpdate`.

You can nest `BeginUpdate` and `EndUpdate` calls in assembly language if you save and restore the copy of the `visRgn`, a copy of which is stored, in global coordinates, in the global variable `SaveVisRgn`. ◆

SPECIAL CONSIDERATIONS

If you don't clear the update region when you receive an update event, the Event Manager continues to send update events until you do.

SEE ALSO

See Figure 4-21 on page 4-49 for an illustration of how `BeginUpdate` and `EndUpdate` affect the visible region and update region. See Listing 4-10 on page 4-50 for an example that updates a window.

EndUpdate

Use the `EndUpdate` procedure to finish updating a window.

```
PROCEDURE EndUpdate (theWindow: WindowPtr);
```

`theWindow` A pointer to the window's window record.

DESCRIPTION

The `EndUpdate` procedure restores the normal visible region of a window's graphics port. When you receive an update event for a window, you call `BeginUpdate`, redraw the update region, and then call `EndUpdate`. Each call to `BeginUpdate` must be balanced by a subsequent call to `EndUpdate`.

SEE ALSO

See Figure 4-21 on page 4-49 for an illustration of how `BeginUpdate` and `EndUpdate` affect the visible region and update region. See Listing 4-10 on page 4-50 for an example that updates a window.

InvalRect

Use the `InvalRect` procedure to add a rectangle to a window's update region.

```
PROCEDURE InvalRect (badRect: Rect);
```

`badRect` A rectangle, in local coordinates, that is to be added to a window's update region.

DESCRIPTION

The `InvalRect` procedure adds a specified rectangle to the update region of the window whose graphics port is the current port. Specify the rectangle in local coordinates. The Window Manager clips it, if necessary, to fit in the window's content region.

Both your application and the Window Manager use the `InvalRect` procedure. When the user enlarges a window, for example, the Window Manager uses `InvalRect` to add the newly created content region to the update region. Your application uses `InvalRect` to add the two rectangles formerly occupied by the scroll bars in the smaller content area.

InvalRgn

Use the `InvalRgn` procedure to add a region to a window's update region.

```
PROCEDURE InvalRgn (badRgn: RgnHandle);
```

`badRgn` The region, in local coordinates, that is to be added to a window's update region.

DESCRIPTION

The `InvalRgn` procedure adds a specified region to the update region of the window whose graphics port is the current port. Specify the region in local coordinates. The Window Manager clips it, if necessary, to fit in the window's content region.

SEE ALSO

See Listing 4-13 on page 4-58 for an example that uses `InvalRgn` to add part of the window's content region to the update region.

ValidRect

Use the `ValidRect` procedure to remove a rectangle from a window's update region.

```
PROCEDURE ValidRect (goodRect: Rect);
```

`goodRect` A rectangle, in local coordinates, to be removed from a window's update region.

DESCRIPTION

The `ValidRect` procedure removes a specified rectangle from the update region of the window whose graphics port is the current port. Specify the region in local coordinates. The Window Manager clips it, if necessary, to fit in the window's content region.

Your application uses `ValidRect` to tell the Window Manager that it has already drawn a rectangle and to cancel any updates accumulated for that area. You can thereby improve response time by reducing redundant redrawing.

Suppose, for example, that you've resized a window that contains a size box and scroll bars. Depending on the dimensions of the newly sized window, the new size box and scroll bar areas may or may not have been accumulated into the window's update region. After calling `SizeWindow`, you can redraw the size box or scroll bars immediately and then call `ValidRect` for the areas they occupy. If they were in fact accumulated into the update region, `ValidRect` removes them so that you do not have to redraw them with the next update event.

SEE ALSO

See Listing 4-13 on page 4-58 for an example that uses `ValidRect` to remove part of the window's content region from the update region.

ValidRgn

Use the `ValidRgn` procedure to remove a specified region from a window's update region.

```
PROCEDURE ValidRgn (goodRgn: RgnHandle);
```

`goodRgn` A region, in local coordinates, to be removed from a window's update region.

DESCRIPTION

The `ValidRgn` procedure removes a specified region from the update region of the window whose graphics port is the current port. Specify the region in local coordinates. The Window Manager clips it, if necessary, to fit in the window's content region.

Setting and Retrieving Other Window Characteristics

This section describes the routines that let you set and retrieve less commonly used fields in the window record.

SetWindowPic

Use the `SetWindowPic` procedure to establish a picture that the Window Manager can draw in a window's content region.

```
PROCEDURE SetWindowPic (theWindow: WindowPtr;
                        Pic: PicHandle);
```

`theWindow` A pointer to a window's window record.

`Pic` A handle to the picture to be drawn in the window.

DESCRIPTION

The `SetWindowPic` procedure stores in a window's window record a handle to a picture to be drawn in the window. When the window's content region must be updated, the Window Manager then draws the picture or part of the picture, as necessary, instead of generating an update event.

Note
The `CloseWindow` and `DisposeWindow` procedures assume that any picture pointed to by the window record field `windowPic` is stored as data, not as a resource. If your application uses a picture stored as a resource, you must release the memory it occupies by calling the Resource Manager's `ReleaseResource` procedure and set the `WindowPic` field to `NIL` before you close the window. ◆

GetWindowPic

Use the `GetWindowPic` function to retrieve a handle to a window's picture.

```
FUNCTION GetWindowPic (theWindow: WindowPtr): PicHandle;
```

`theWindow` A pointer to the window's window record.

DESCRIPTION

The `GetWindowPic` function returns a handle to the picture to be drawn in a specified window's content region. The handle must have been stored previously with the `SetWindowPic` procedure.

SetWRefCon

Use the `SetWRefCon` procedure to set the `refCon` field of a window record.

```
PROCEDURE SetWRefCon (theWindow: WindowPtr; data: LongInt);
```

`theWindow` A pointer to the window's window record.
`data` The data to be placed in the `refCon` field.

DESCRIPTION

The `SetWRefCon` procedure places the specified data in the `refCon` field of the specified window record. The `refCon` field is available to your application for any window-related data it needs to store.

SEE ALSO

See Listing 4-3 on page 4-28 for an example that sets the `refCon` field. See Listing 4-16 on page 4-60 for an example that uses the contents of the `refCon` field.

GetWRefCon

Use the `GetWRefCon` function to retrieve the reference constant from a window's window record.

```
FUNCTION GetWRefCon (theWindow: WindowPtr): LongInt;
```

`theWindow` A pointer to the window's window record.

DESCRIPTION

The `GetWRefCon` function returns the long integer data stored in the `refCon` field of the specified window record.

SEE ALSO

See the section "Managing Multiple Windows" beginning on page 4-23 for suggested ways to use the `refCon` field. See Listing 4-1 on page 4-25 for an example of an application-defined routine that gets the `refCon` field.

GetWVariant

Use the `GetWVariant` function to retrieve a window's variation code.

```
FUNCTION GetWVariant (theWindow: WindowPtr): Integer;
```

`theWindow` A pointer to the window's window record.

DESCRIPTION

The `GetWVariant` function returns the variation code of the specified window. Depending on the window's window definition function, the result of `GetWVariant` can represent one of the standard window types listed in the section "Creating a Window" beginning on page 4-25 or a variation code defined by your own window definition function.

SEE ALSO

See "Types of Windows" beginning on page 4-8 for a definition of variation codes. See "The Window Definition Function" beginning on page 4-120 for a detailed description of variation codes.

Manipulating the Desktop

This section describes the routines that let your application retrieve information about the desktop and set the desktop pattern. Ordinarily, your application doesn't need to manipulate any part of the desktop outside of its own windows.

SetDeskCPat

Use the `SetDeskCPat` procedure to set the desktop pattern on a computer that supports Color QuickDraw.

```
PROCEDURE SetDeskCPat (deskPixPat: PixPatHandle);
```

`deskPixPat` A handle to a pixel pattern.

DESCRIPTION

The `SetDeskCPat` procedure sets the desktop pattern to a specified pixel pattern, which can be drawn in more than two colors. After a call to `SetDeskCPat`, the desktop is automatically redrawn in the new pattern. If the specified pattern is a binary pattern (with a pattern type of 0), it is drawn is the current foreground and background colors. If the value of the `deskPixPat` parameter is `NIL`, `SetDeskCPat` uses the standard binary desk pattern (that is, the `'ppat'` resource with resource ID 16).

Note
For compatibility with other Macintosh applications and the system software, applications should ordinarily not change the desktop pattern. ◆

The Window Manager's desktop-painting routines can paint the desktop either in the binary pattern stored in the global variable `DeskPattern` or in a new pixel pattern. The desktop pattern used at startup is determined by the value of the parameter-RAM bit flag called `pCDeskPat`. If the value of `pCDeskPat` is 0, the Window Manager uses the new pixel pattern; if not, it uses the binary pattern stored in `DeskPattern`. The user can change the color pattern through the General Controls panel, which changes the value of `pCDeskPat`.

GetGrayRgn

Use the `GetGrayRgn` function to retrieve a handle to the current desktop region.

```
FUNCTION GetGrayRgn: RgnHandle;
```

DESCRIPTION

The `GetGrayRgn` function returns a handle to the current desktop region from the global variable `GrayRgn`.

The desktop region represents all available screen space, that is, the desktop area displayed by all monitors attached to the computer. Ordinarily, your application doesn't need to access the desktop region directly.

When your application calls `DragWindow` to let the user drag a window, it can use `GetGrayRgn` to set the limiting rectangle to the entire desktop area.

SEE ALSO

See Listing 4-9 on page 4-44 for an example that uses `GetGrayRgn` to specify the limiting rectangle when calling `DragWindow` to let the user move a window.

GetCWMgrPort

Use the `GetCWMgrPort` procedure to retrieve a pointer to the Window Manager port on a system that supports Color QuickDraw.

```
PROCEDURE GetCWMgrPort (VAR wMgrCPort: CGrafPtr);
```

`wMgrCPort` A parameter in which `GetCWMgrPort` returns a pointer to the Window Manager port.

DESCRIPTION

The `GetCWMgrPort` procedure places a pointer to the color Window Manager port in the parameter `wMgrCPort`. The `GetCWMgrPort` procedure is available only on computers with Color QuickDraw.

The Window Manager port is a graphics port that occupies all of the main screen. Ordinarily, your application doesn't need to access the Window Manager port.

Note
Do not change any regions of the Window Manager port. If you do, the Window Manager might not handle overlapping windows properly. ◆

GetWMgrPort

Use the `GetWMgrPort` procedure to retrieve a pointer to the Window Manager port on a system with only the original monochrome QuickDraw.

```
PROCEDURE GetWMgrPort (VAR wPort: GrafPtr);
```

`wPort` A parameter in which `GetWMgrPort` returns a pointer to the Window Manager port.

DESCRIPTION

The `GetWMgrPort` procedure places a pointer to the Window Manager port in the parameter `wPort`.

The Window Manager port is a graphics port that occupies all of the main screen. Ordinarily, your application doesn't need to access the Window Manager port.

Note
Do not change any regions of the Window Manager port. If you do, the Window Manager might not handle overlapping windows properly. ◆

Manipulating Window Color Information

This section describes the routines you use for setting and retrieving window color information. Your application does not normally change window color information.

SetWinColor

Use the `SetWinColor` procedure to set a window's window color table.

```
PROCEDURE SetWinColor (theWindow: WindowPtr;
                       newColorTable: WCTabHandle);
```

`theWindow` A pointer to the window's window record.

`newColorTable`
A handle to a window color table record, which defines the colors for the window's new color table.

DESCRIPTION

The `SetWinColor` procedure sets a window's color table. If the window has no auxiliary window record, it creates a new one with the specified window color table and adds it to the auxiliary window list. If the window already has an auxiliary record, its window color table is replaced. The Window Manager then redraws the window frame and highlighted text in the new colors and sets the window's background color to the new content color.

If the new color table has the same entries as the default color table, `SetWinColor` changes the auxiliary window record so that it points to the default color table.

Window color table resources (resources of type `'wctb'`) should not be purgeable.

If you specify a value of `NIL` for the parameter `theWindow`, `SetWinColor` changes the default color table in memory. Your application shouldn't, however, change the default color table.

SEE ALSO

For a description of a window color table, see "The Window Color Table Record" on page 4-71. For a description of the auxiliary window record, see "The Auxiliary Window Record" on page 4-73. For a description of the `'wctb'` resource, see "The Window Color Table Resource" on page 4-127.

GetAuxWin

Use the `GetAuxWin` function to retrieve a handle to a window's auxiliary window record.

```
FUNCTION GetAuxWin (theWindow: WindowPtr;
                    VAR awHndl: AuxWinHandle): Boolean;
```

`theWindow` A pointer to the window's window record.

`awHndl` A handle to the window's auxiliary window record.

DESCRIPTION

The `GetAuxWin` function returns a Boolean value that reports whether or not the window has an auxiliary window record, and it sets the variable parameter `awHndl` to the window's auxiliary window record.

If the window has no auxiliary window record, `GetAuxWin` places the default window color table in `awHndl` and returns a value of `FALSE`.

SEE ALSO

For a description of the auxiliary window record, see "The Auxiliary Window Record" on page 4-73.

Low-Level Routines

This section describes the low-level routines that are called by higher-level Window Manager routines. Ordinarily, you won't need to use these routines.

CheckUpdate

The Event Manager uses the `CheckUpdate` function to scan the window list for windows that need updating.

```
FUNCTION CheckUpdate (VAR theEvent: EventRecord): Boolean;
```

`theEvent` An event record to be filled in if a window needs updating.

DESCRIPTION

The `CheckUpdate` function scans the window list from front to back, checking for a visible window that needs updating (that is, a visible window whose update region is not empty). If it finds one whose window record contains a picture handle, it redraws the window itself and continues through the list. If it finds a window record whose update region is not empty and whose window record does not contain a picture handle, it stores an update event in the parameter `theEvent` and returns `TRUE`. If it finds no such window, it returns `FALSE`.

The Event Manager is the only software that ordinarily calls `CheckUpdate`.

ClipAbove

The Window Manager uses the `ClipAbove` procedure to determine the clip region of the Window Manager port for displaying a window.

```
PROCEDURE ClipAbove (window: WindowPeek);
```

`window` A pointer to the window's complete window record.

DESCRIPTION

The `ClipAbove` procedure sets the clip region of the Window Manager port to be the area of the desktop that intersects the current clip region, minus the structure regions of all the windows in front of the specified window.

The `ClipAbove` procedure retrieves the desktop region from the global variable `GrayRgn`.

SaveOld

The Window Manager uses the `SaveOld` procedure to save a window's current structure and content regions preparatory to updating the window.

```
PROCEDURE SaveOld (window: WindowPeek);
```

`window` A pointer to the window's complete window record.

DESCRIPTION

The `SaveOld` procedure saves the specified window's current structure region and content region for the `DrawNew` procedure. Each call to `SaveOld` must be balanced by a subsequent call to `DrawNew`.

DrawNew

The Window Manager uses the `DrawNew` procedure to erase and update changed window regions.

```
PROCEDURE DrawNew (window: WindowPeek; update: Boolean);
```

`window` A pointer to the window's complete window record.

`update` A Boolean value that determines whether the regions are updated.

DESCRIPTION

The `DrawNew` procedure erases the parts of a window's structure and content regions that are part of the window's former state and part of its new state but not both. That is,

```
(OldStructure XOR NewStructure) UNION (OldContent XOR NewContent)
```

If the update parameter is set to `TRUE`, `DrawNew` also updates the erased regions.

▲ **WARNING**
In Pascal, `SaveOld` and `DrawNew` are not nestable. ▲

ASSEMBLY-LANGUAGE INFORMATION

In assembly language, you can nest `SaveOld` and `DrawNew` if you save and restore the values of the global variables `OldStructure` and `OldContent`.

PaintOne

The Window Manager uses the `PaintOne` procedure to redraw the invalid, exposed portions of one window on the desktop.

```
PROCEDURE PaintOne (window: WindowPeek; clobberedRgn: RgnHandle);
```

`window` A pointer to the window's complete window record.

`clobberedRgn`
A handle to the region that has become invalid.

DESCRIPTION

The `PaintOne` procedure "paints" the invalid portion of the specified window and all windows above it. It draws as much of the window frame as is in `clobberedRgn` and, if some content region is exposed, erases the exposed area (paints it with the background pattern) and adds it to the window's update region. If the value of the `window` parameter is `NIL`, the window is the desktop, and `PaintOne` paints it with the desktop pattern.

ASSEMBLY-LANGUAGE INFORMATION

The global variables `SaveUpdate` and `PaintWhite` are flags used by `PaintOne`. Normally both flags are set. Clearing `SaveUpdate` prevents `clobberedRgn` from being added to the window's update region. Clearing `PaintWhite` prevents `clobberedRgn` from being erased before being added to the update region (this is useful, for example, if the background pattern of the window isn't the background pattern of the desktop). The Window Manager sets both flags periodically, so you should clear the appropriate flags each time you need them to be clear.

PaintBehind

The Window Manager uses the `PaintBehind` procedure to redraw a series of windows in the window list.

```
PROCEDURE PaintBehind (startWindow: WindowPeek;
                       clobberedRgn: RgnHandle);
```

`startWindow`
A pointer to the window's complete window record.

`clobberedRgn`
A handle to the region that has become invalid.

DESCRIPTION

The `PaintBehind` procedure calls `PaintOne` for `startWindow` and all the windows behind `startWindow`, clipped to `clobberedRgn`.

ASSEMBLY-LANGUAGE INFORMATION

Because `PaintBehind` clears the global variable `PaintWhite` before calling `PaintOne`, `clobberedRgn` isn't erased. The `PaintWhite` global variable is reset after the call to `PaintOne`.

CalcVis

The Window Manager uses the `CalcVis` procedure to calculate the visible region of a window.

```
PROCEDURE CalcVis (window: WindowPeek);
```

`window` A pointer to the window's complete window record.

DESCRIPTION

The `CalcVis` procedure calculates the visible region of the specified window by starting with its content region and subtracting the structure region of each window in front of it.

CalcVisBehind

The Window Manager uses the `CalcVisBehind` procedure to calculate the visible regions of a series of windows.

```
PROCEDURE CalcVisBehind (startWindow: WindowPeek;
                         clobberedRgn: RgnHandle);
```

`startWindow`
A pointer to a window's window record.

`clobberedRgn`
A handle to the desktop region that has become invalid.

DESCRIPTION

The `CalcVisBehind` procedure calculates the visible regions of the window specified by the `startWindow` parameter and all windows behind `startWindow` that intersect `clobberedRgn`. It is called after `PaintBehind`.

Application-Defined Routine

This section describes the window definition function. The Window Manager supplies window definition functions that handle the standard window types described in "Types of Windows" beginning on page 4-8.

The Window Definition Function

If your application defines its own window types, you must supply your own window definition function to handle them. Store your definition function as a resource of type `'WDEF'` with an ID from 128 through 4096. (Window definition function resource IDs 0 and 1 are the default window definition functions; resource IDs 2 through 127 are reserved by Apple Computer, Inc.)

Your window definition function can support up to 16 variation codes, which are identified by integers 0 through 15. To invoke your own window type, you specify the window's definition ID, which contains the resource ID of the window's definition function in the upper 12 bits and the variation code in the lower 4 bits. Thus, for a given resource ID and variation code, the window definition ID is

(16 * resource ID) + (variation code)

When you create a window, the Window Manager calls the Resource Manager to access the window definition function. The Resource Manager reads the window definition function into memory and returns a handle to it. The Window Manager stores this handle in the `windowDefProc` field of the window record. (If 24-bit addressing is in effect, the Window Manager stores the variation code in the lower 4 bits of the `windowDefProc` field; if 32-bit addressing is in effect, the Window Manager stores the variation code elsewhere.) Later, when it needs to perform a type-dependent action on the window, the Window Manager calls the window definition function and passes it the variation code as a parameter.

MyWindow

The window definition function is responsible for drawing the window frame, reporting the region where mouse-down events occur, calculating the window's structure region and content region, drawing the size box, resizing the window frame when the user drags the size box, and performing any customized initialization or disposal tasks.

You can give your window definition function any name you wish. It takes four parameters and returns a result code:

```
FUNCTION MyWindow (varCode: Integer; theWindow: WindowPtr;
                   message: Integer; param: LongInt): LongInt;
```

`varCode` The window's variation code.

`theWindow` A pointer to the window's window record.

`message` A code for the task to be performed. The `message` parameter has one of these values:

```
CONST
   wDraw       = 0;  {draw window frame}
   wHit        = 1;  {report where mouse-down event }
                     { occurred}
   wCalcRgns   = 2;  {calculate strucRgn and contRgn}
   wNew        = 3;  {perform additional }
                     { initialization}
   wDispose    = 4;  {perform additional disposal }
                     { tasks}
   wGrow       = 5;  {draw grow image during resizing}
   wDrawGIcon  = 6;  {draw size box and scroll bar }
                     { outline}
```

The subsections that follow explain each of these tasks in detail.

`param` Data associated with the task specified by the `message` parameter. If the task requires no data, this parameter is ignored.

Your window definition function performs whatever task is specified by the `message` parameter and returns a function result if appropriate. If the task performed requires no result code, return 0.

The function's entry point must be at the beginning of the function.

You can set up the various tasks as subroutines inside the window definition function, but you're not required to do so.

Drawing the Window Frame

When you receive a `wDraw` message, draw the window frame in the current graphics port, which is the Window Manager port.

You must make certain checks to determine exactly how to draw the frame. If the value of the `visible` field in the window record is `FALSE`, you should do nothing; otherwise, you should examine the `param` parameter and the status flags in the window record:

- If the value of `param` is 0, draw the entire window frame.
- If the value of `param` is 0 and the `hilited` field in the window record is `TRUE`, highlight the frame to show that the window is active.
  - If the value of the `goAwayFlag` field in the window record is also `TRUE`, draw a close box in the window frame.
  - If the value of the `spareFlag` field in the window record is also `TRUE`, draw a zoom box in the window frame.
- If the value of the `param` parameter is `wInGoAway`, add highlighting to, or remove it from, the window's close box. Figure 4-19 on page 4-46 illustrates the close box with and without highlighting as drawn by the Window Manager's window definition function.

- If the value of the `param` parameter is `wInZoom`, add highlighting to, or remove it from, the window's zoom box. Figure 4-20 on page 4-47 illustrates the zoom box with and without highlighting as drawn by the Window Manager's window definition function.

Note
When the Window Manager calls a window definition function with a message of `wDraw`, it stores a value of type `Integer` in the `param` parameter without clearing the high-order word. When processing the `wDraw` message, use only the low-order word of the `param` parameter. ◆

The window frame typically but not necessarily includes the window's title, which should be displayed in the system font and system font size. The Window Manager port is already set to use the system font and system font size.

When designing a title bar that includes the window title, allow at least 16 pixels vertically to support localization for script systems in which the system font can be no smaller than 12 points.

Note
Nothing drawn outside the window's structure region is visible. ◆

Returning the Region of a Mouse-Down Event

When you receive a `wHit` message, you must determine where the cursor was when the mouse button was pressed. The `wHit` message is accompanied by the mouse location, in global coordinates, in the `param` parameter. The vertical coordinate is in the high-order word of the parameter, and the horizontal coordinate is in the low-order word. You return one of these constants:

```
CONST
   wNoHit      = 0;  {none of the following}
   wInContent  = 1;  {in content region (except grow, if active)}
   wInDrag     = 2;  {in drag region}
   wInGrow     = 3;  {in grow region (active window only)}
   wInGoAway   = 4;  {in go-away region (active window only)}
   wInZoomIn   = 5;  {in zoom box for zooming in (active window }
                     { only)}
   wInZoomOut  = 6;  {in zoom box for zooming out (active window }
                     { only)}
```

The return value `wNoHit` might mean (but not necessarily) that the point isn't in the window. The standard window definition functions, for example, return `wNoHit` if the point is in the window frame but not in the title bar.

Return the constants `wInGrow`, `wInGoAway`, `wInZoomIn`, and `wInZoomOut` only if the window is active—by convention, the size box, close box, and zoom box aren't drawn if the window is inactive. In an inactive document window, for example, a mouse-down event in the part of the title bar that would contain the close box if the window were active is reported as `wInDrag`.

Calculating Regions

When you receive the `wCalcRgns` message, you calculate the window's structure and content regions based on the current graphics port's port rectangle. These regions, whose handles are in the `strucRgn` and `contRgn` fields of the window record, are in global coordinates. The Window Manager requests this operation only if the window is visible.

▲ **WARNING**
When you calculate regions for your own type of window, do not alter the clip region or the visible region of the window's graphics port. The Window Manager and QuickDraw take care of this for you. Altering the clip region or visible region may damage other windows. ▲

Initializing a New Window

When you receive the `wNew` message, you can perform any type-specific initialization that may be required. If the content region has an unusual shape, for example, you might allocate memory for the region and store the region handle in the `dataHandle` field of the window record. The initialization routine for a standard document window creates the `wStateData` record for storing zooming data.

Disposing of a Window

When you receive the `wDispose` message, you can perform any additional tasks necessary for disposing of a window. You might, for example, release memory that was allocated by the initialization routine. The dispose routine for a standard document window disposes of the `wStateData` record.

Resizing a Window

When you receive the `wGrow` message, draw a grow image of the window. With the `wGrow` message you receive a pointer to a rectangle, in global coordinates, whose upper-left corner is aligned with the port rectangle of the window's graphics port. Your grow image should fit inside the rectangle. As the user drags the mouse, the Window Manager sends repeated `wGrow` messages, so that you can change your grow image to match the changing mouse location.

Draw the grow image in the current graphics port, which is the Window Manager port, in the current pen pattern and pen mode. These are set up (as `gray` and `notPatXor`) to conform to the Macintosh user interface guidelines.

The grow routine for a standard document window draws a dotted (gray) outline of the window and also the lines delimiting the title bar, size box, and scroll bar areas.

Drawing the Size Box

When you receive the `wDrawGIcon` message, you draw the size box in the content region if the window is active—if the window is inactive, draw whatever is appropriate to show that the window cannot currently be sized.

Note
If the size box is located in the window frame instead of the content region, do nothing in response to the `wDrawGIcon` message, instead drawing the size box in response to the `wDraw` message. ◆

The routine that draws a size box for an active document window draws the size box in the lower-right corner of the port rectangle of the window's graphics port. It also draws lines delimiting the size box and scroll bar areas. For an inactive document window, it erases the size box and draws the delimiting lines.

Resources

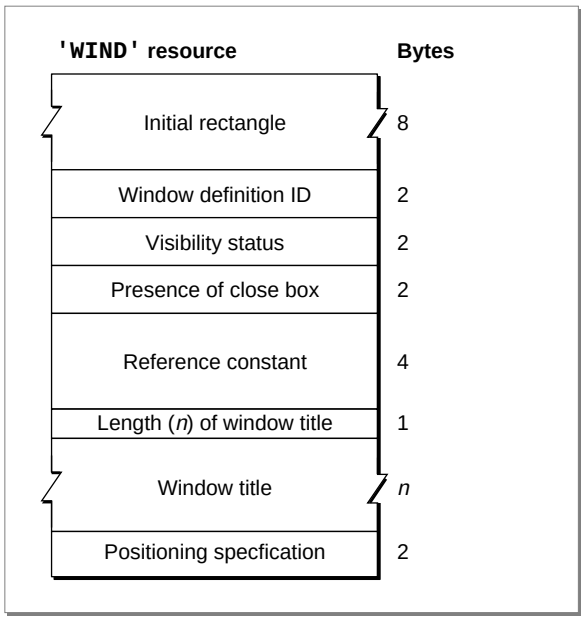
This section describes the resources used by the Window Manager:

- the `'WIND'` resource, used for describing the characteristics of windows
- the `'WDEF'` resource, which holds a window definition function
- the `'wctb'` resource, which defines the colors to be used for a window's frame and highlighting

The Window Resource

You typically define a window resource for each type of window that your application creates. Figure 4-24 illustrates a compiled `'WIND'` resource.

Figure 4-24 Structure of a compiled window (`'WIND'`) resource



A compiled version of a window resource contains the vollowing elements:

- The upper-left and lower-right corners, in global coordinates, of a rectangle that defines the initial size and placement of the window's content region. Your application can change this rectangle before displaying the window, either programmatically or through an optional positioning code described later in this section.
- The window's definition ID, which incorporates both the resource ID of the window definition function that will handle the window and an optional variation code. Together, the window definition function resource ID and the variation code define a window type. Place the resource ID of the window definition function in the upper 12 bits of the definition ID. Window definition functions with IDs 0 through 127 are reserved for use by Apple Computer, Inc. Place the optional variation code in the lower 4 bits of the definition ID.

  If you're using one of the standard window types (described in "Types of Windows" beginning on page 4-8), the definition ID is one of the window-type constants:

```
CONST
documentProc      = 0;  {movable, sizable window, }
                        { no zoom box}
dBoxProc          = 1;  {alert box or modal dialog box}
plainDBox         = 2;  {plain box}
altDBoxProc       = 3;  {plain box with shadow}
noGrowDocProc     = 4;  {movable window, no size box or }
                        { zoom box}
movableDBoxProc   = 5;  {movable modal dialog box}
zoomDocProc       = 8;  {standard document window}
zoomNoGrow        = 12; {zoomable, nonresizable window}
rDocProc          = 16; {rounded-corner window}
```

  You can also add a zoom box to a movable modal dialog box by specifying the sum of two constants: `movableDBoxProc + zoomDocProc`, but a zoom box is not recommended on any dialog box.

  You can control the angle of curvature on a rounded-corner window (window type `rDocProc`) by adding one of these integers:

Window definition ID	Diameters of curvature
`rDocProc`	16, 16
`rDocProc + 2`	4, 4
`rDocProc + 4`	6, 6
`rDocProc + 6`	10, 10

- A specification that determines whether the window is visible or invisible. This characteristic controls only whether the Window Manager displays the window, not necessarily whether the window can be seen on the screen. (A visible window entirely covered by other windows, for example, is "visible" even though the user cannot see it.) You typically create a new window in an invisible state, build the content area of the window, and then display the completed window.

- A specification that determines whether or not the window has a close box. The Window Manager draws the close box when it draws the window frame. The window type specified in the second field determines whether a window can support a close box; this field determines whether the close box is present.
- A reference constant, which your application can use for whatever data it needs to store. When it builds a new window record, the Window Manager stores, in the `refCon` field, whatever value you specify in the fifth element of the window resource. You can also put a placeholder here and then set the `refCon` field yourself with the `SetWRefCon` procedure.
- A string that specifies the window title. The first byte of the string specifies the length of the string (that is, the number of characters in the title plus 1 byte for the length), in bytes.
- An optional positioning specification that overrides the window position established by the rectangle in the first field. The positioning value can be one of the integers defined by the constants listed here. In these constant names, the terms have the following meanings:

`center`	Centered both horizontally and vertically, relative either to a screen or to another window (if a window to be centered relative to another window is wider than the window that preceded it, it is pinned to the left edge; a narrower window is centered)
`stagger`	Located 10 pixels to the right and 10 pixels below the upper-left corner of the last window (in the case of staggering relative to a screen, the first window is placed just below the menu bar at the left edge of the screen, and subsequent windows are placed on that screen relative to the first window)
`alert position`	Centered horizontally and placed in the "alert position" vertically, that is, with about one-fifth of the window or screen above the new window and the rest below
`parent window`	The window in which the user was last working

The seventh element of the resource can contain one of the values specified by these constants:

```
CONST noAutoCenter            = 0x0000;{use initial }
                                       { location}
      centerMainScreen        = 0x280A;{center 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;{place in alert }
                                    { position on }
                                    { parent window }
                                    { screen}
staggerParentWindowScreen  = 0x780A;{stagger on parent }
                                    { window screen}
```

The positioning constants are convenient when the user is creating new documents or when you are handling your own dialog boxes and alert boxes. When you are creating a new window to display a previously saved document, however, you should display the new window in the same rectangle as the previous window (that is, the window the document occupied when it was last saved). For more information, see "Positioning a Document Window on the Desktop" beginning on page 4-30.

Use the `GetNewCWindow` or `GetNewWindow` function to read a `'WIND'` resource. Both functions create a new window record and fill it in according to the values specified in a `'WIND'` resource.

The Window Definition Function Resource

Window definition functions are stored as resources of type `'WDEF'`. The `'WDEF'` resource is simply the executable code for the window definition function.

The two standard window definition functions supplied by the Window Manager use resource IDs 0 and 1.

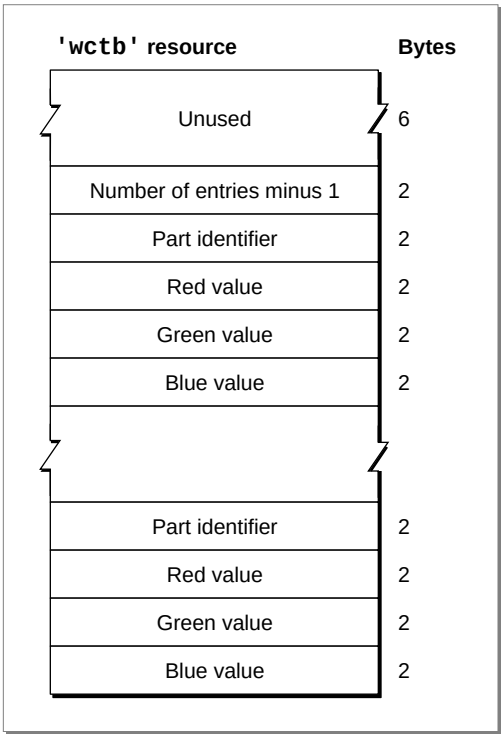
The Window Color Table Resource

You can specify your own window color tables as resources of type `'wctb'`.

Ordinarily, you should not define your own window color tables, unless you have some extraordinary need to control the color of a window's frame or text highlighting. To assign a table to a window when you create the window, provide a window color table (`'wctb'`) resource with the same resource ID as the `'WIND'` resource from which you create the window.

The window color table resource is an exact image of the window color table data structure. Figure 4-25 illustrates the contents of a compiled `'wctb'` resource.

Figure 4-25 Structure of a compiled window color table (`'wctb'`) resource



A compiled version of a window resource contains the following elements:

- An unused field 6 bytes long.
- An integer that specifies the number of entries in the resource (that is, the number of color specification records) minus 1.
- A series of color specification records, each of which consists of a 2-byte part identifier and three 2-byte color values. The part identifier is an integer specified by one of these constants:

```
CONST wContentColor     = 0;  {content region background}
      wFrameColor       = 1;  {window frame}
      wTextColor        = 2;  {window title and button text}
      wHiliteColor      = 3;  {reserved}
      wTitleBarColor    = 4;  {reserved}
      wHiliteColorLight = 5;  {lightest stripes in title bar }
                              { and lightest dimmed text}
      wHiliteColorDark  = 6;  {darkest stripes in title bar }
                              { and darkest dimmed text}
      wTitleBarLight    = 7;  {lightest parts of title bar }
                              { background}
```

```
wTitleBarDark     = 8;  {darkest parts of title bar }
                        { background}
wDialogLight      = 9;  {lightest element of dialog box }
                        { frame}
wDialogDark       = 10; {darkest element of dialog box }
                        { frame}
wTingeLight       = 11; {lightest window tinging}
wTingeDark        = 12; {darkest window tinging}
```

The color values are simply the intensity of the red, green, and blue in each window part (see *Inside Macintosh: Imaging* for a description of RGB color).