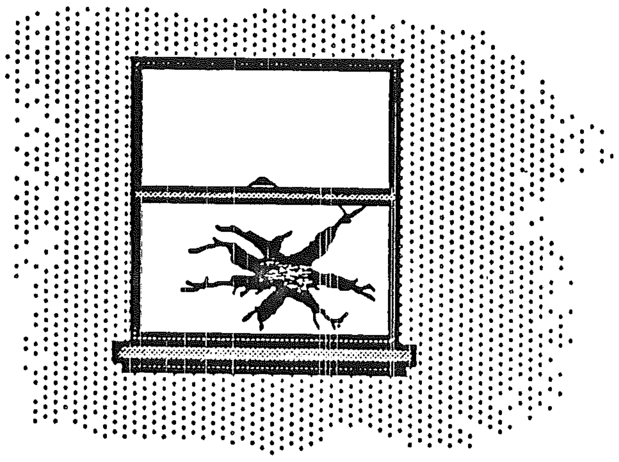
Cortland Window Manager

Dan Oliver

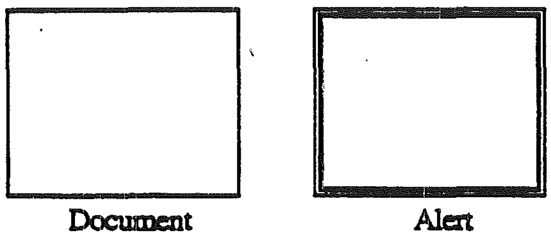


01/30/86	Initial release
06/05/86	Revised release. Note the removal of direct access to window records by applications and additional calls to compensate. The inputs to **NewWindow** has become more Mac like, and **DisposeWindow** is folded into **CloseWindow**.
06/10/86	Updates to page 4, and page 6 in the appendix.
06/14/86	Updates to pages 15-20.
06/18/86	Name changes to **BootWmgr**, **InitWindows**, **TermWindows**, and **WmgrVersion**. Addition of **WindReset** and **WindStatus**, although not completed. Additional parameter, user ID, passed to **WindStartup** (formerly **InitWindows**). **TaskMaster** uses an extended event record.
07/15/86	Expanded **SetFrameColor**. **WNewRes** doesn't redraw the screen anymore. Changes to color table in "WINDOW FRAME COLORS AND PATTERNS". New input to **NewWindow**. New calls; **GetCOrgin, SetCOrigin, GetDataSize, SetDataSize, GetMaxGrow, SetMaxGrow, GetScroll, SetScroll, GetPage, SetPage, GetCDraw, SetCDraw, GetInfoDraw, SetInfoDraw, StartDrawing**. New sections DRAW CONTENT ROUTINE and DRAW INFORMATION BAR ROUTINE. New input parameters to **MoveWindow**. Defaults added to **DragWindow**. Parameters expressed as "POINT" are now broken down into two WORDs.
07/16/86	Replacement for pages 6-8 in the appendix which lables **NewWindow** parameter list. Insert pages 15.a-15.c between pages 15 and 16. These pages define DRAW CONTENT ROUTINE and DRAW INFORMATION BAR ROUTINE as promised in the last release.
08/13/86	Two bits added to wframe field of window record (see **NewWindow**). **SetOrgnMask** call added for scrollable windows that use color dithering in 640 mode. Parameter length field added to parameter list passed to **NewWindow**. **SetWMgrIcons** call added along with a WINDOW MANAGER ICON FONT section. **TaskMaster** returns window pointer in TaskData field rather than the message field.

ABOUT THE WINDOW MANAGER

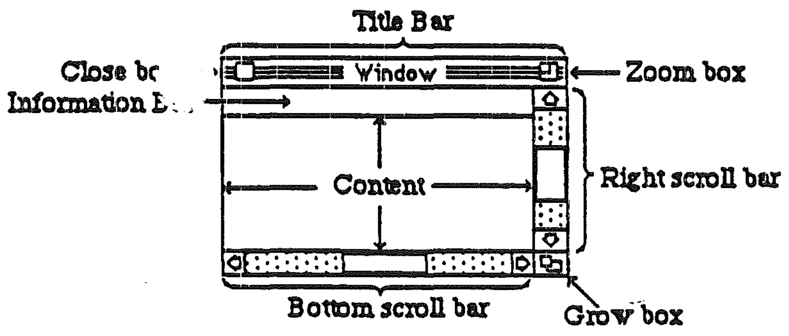
The Window Manager is a tool for dealing with windows on the Cortland screen. The screen represents a working surface or **desktop**; graphic objects appear on the desktop and can be manipulated with a mouse. A **window** is an object on the desktop that presents information, such as a document or a message. Windows can be any size or shape, and there can be one or many of them, depending an the application.

There are two kinds of predefined window frames, **document** and **alert**.



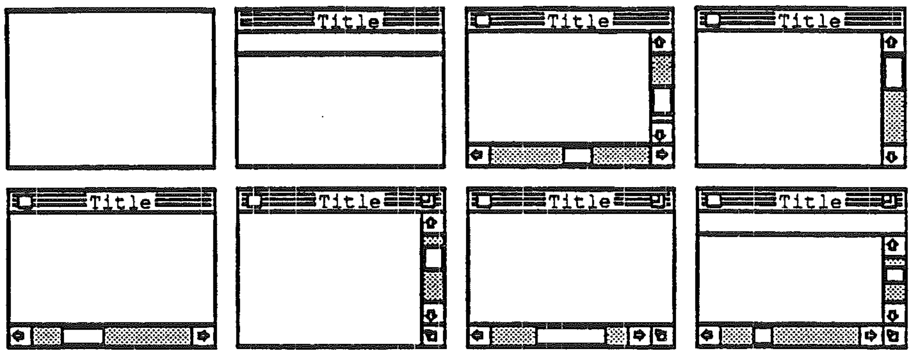
The alert window is used by the Dialog Manager and is explained in that ERS.

Inside the document window can be **standard window controls**, which are; title bar, close box, zoom box, right scroll bar, bottom scroll bar, grow box, and information bar. The title bar displays the window's title, can hold the close and zoom boxes, and can be a drag region for moving the window. The close box is selected by the user to remove the window from the screen. The zoom box is selected by the user to make the window its maximum size and to return it to its previous size and position. The right scroll bar is used to scroll vertically through the data in the window. The bottom scroll bar is used to scroll horizontally through the data in the window. The grow box is dragged by the user to change the size of the window. The information bar is a place an application can display some information that won't be effected by the scroll bars.



A document window may have any or all of the standard window controls. The only restriction is that if there is a close or zoom box there must also be a title bar. Common sense would dictate that there only be a zoom box if there is a grow box, although this is not a requirement.

No standard controls may be added to a alert window. Here are some possible document window combinations:



Your application can easily use standard window types, or create your own window types (see DEFINING YOUR OWN WINDOWS). Some windows may be created indirectly for you when you use other parts of the Toolbox; an example is the window the Dialog Manager creates to display an alert. Windows created either directly or indirectly by an application are collectively called **application windows.** There's also a class of windows called **system windows**; these are the windows in which desk accessories are displayed.

The Window Manager's main function is to keep track of overlapping windows. You can draw in any window without running over onto windows in front of it. You can move windows to different places on the screen, change their **plane** (front-to-back order), or change their size, all without concern for how the various windows overlap. The Window Manager kepps track of any newly exposed areas and provides a convenient machanism for you to ensure that they are properly redrawn.

Finally, you can easily set up your application so mouse actions cause these standard responses inside a document window, or similar responses inside other windows:

- Clicking anywhere in an inactive window makes it the active window by bring it to the front and highlighting it.
- Clicking inside the close box of the active window closes the window. Depending on the application, this may mean that the window disappears altogether, or a representation of the window (such as an icon) may be left on the desktop.
- Dragging anywhere inside the title bar of a window (except in the close or zoom boxes, if any) pulls an outline of the window across the screen, and releasing the mouse button moves the window to the new location. If the window isn't the active window, it becomes the active window unless the Command key was also held down. A window can never be moved completely off the screen; by convention, it can't be moved such that the visible area of the title bar is less than four pixels square.
- Dragging inside the size box of the active window changes the size of the window.

WINDOW REGIONS

Every window has the following two regions:

- The **content region:** the area that your application draw in
- The **frame region:** the outline of the entire window plus any standard window controls.

Together, the content and frame regions makeup the **structure region.**

The content region is bounded by the rectangle you specify when you create the window (that is, the portRect of the window's grafPort) The content region is where your application presents information to the user.

A window may also have any of the regions listed below within the window frame.

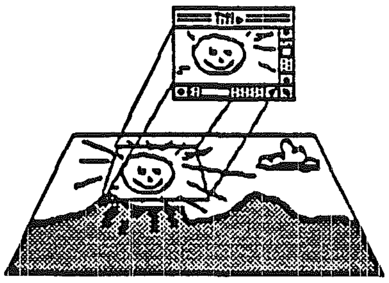
- A **go-away region** , a close box in the active window. Clicking in this region closes the window.
- A **drag region,** the title bar. Dragging in this region pulls an outline of the window across the screen, moves the window to a new location, and makes it the active window (if it isn't already) unless the Command key was held down.
- A **grow region,** the grow box. Dragging in this region pulls the lower right corner of an outline of the window across the screen with the window's origin fixed, resizes the window, and makes it the active window (if it isn't already) unless the Command key was held down.
- A **zoom region,** the zoom box in the active window. Clicking in this region toggles between the current position and size to a maximum size, and back again.

Clicking in any region of an inactive window simply makes it the active window.

> **Note:** The results of clicking and dragging that are discussed here don't happen automatically, unless you are calling TaskMaster; you have to make the right Window Manager calls to cause them to happen.

CONTENT REGION AND WORK AREA

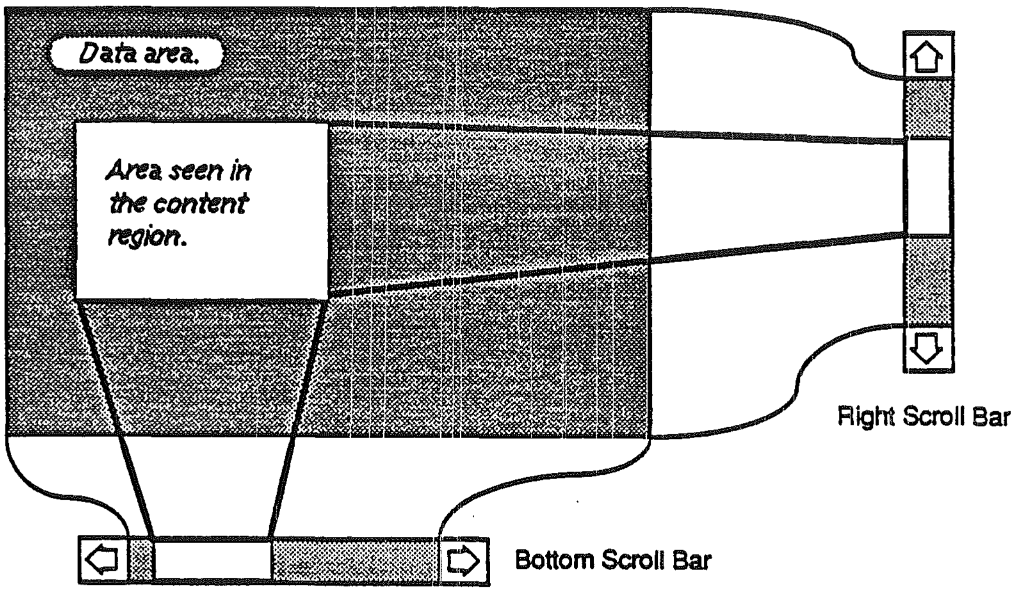
What is the purpose of windows any way? Windows are used to present more information than the hardware (screen) can display at one time, and do it in a standard way. The name window is used because the user sees through the window into a larger area. The power of windows is their ability to give the user a standard device for accessing large amounts of data. Windows act like a microfiche viewer. What is seen on the viewer is like what is seen in the window's content region. And the window's **data area** is what the microfiche is to the viewer. Through the content region the user can see part of the data area, unless the content region is large enough to view the entire data area. Scroll bars are used to scroll the data area through the content region. The grow box and zoom box are used to display more, or less, of the data area at one time. When the window is moved, the data area is moved with it, so the view in the content remains the same.



WINDOW SCROLL BARS

Window scroll bars are the devices used for scrolling the data area through the content region and showing the relationship between the data area and content region. The Control Manager must be installed in order to use scroll bars in windows. Scroll bars are handled by the Control Manager but this document will go over how standard window scroll bars act relating to windows.

The scroll bar is like a reduced cross section of the work area. The scroll thumb is the same ratio to the page region as the content region is to the data area.



USING THE WINDOW MANAGER

To use the Window Manager, you must have previously initialized QuickDraw and the Event Manager. The first Window Manager routine to call is the initialization routine, **WindStartup**.

Where appropriate in your program, use **NewWindow** to create any windows you need.

Now you have a choice to make. There are two ways to handle user input in relation to windows. You can poll the user via **GetNextEvent**, and decide what to do with events, or poll via **TaskMaster**, which will handle most events dealing with standard user interfaces (see USING TASKMASTER).

If you are not using **TaskMaster**, you must poll for events by calling **GetNextEvent** in the Event Manager. For button down events, call **FindWindow**, to see if the button was pressed inside a window. The following are results from **FindWindow** and the standard actions to take:

wInMenuBar	Button passed somewhere outside of the desktop. If you have not subtracted any area from the deskup, there is a good chance it was pressed in the system menu bar. Call **MenuSelect** in the Menu Manager.
wInDrag	Button pressed in a window's drag region, in may not be the active window however. Call **DragWindow**.
wInContent	Button pressed in window's content region. Call **SelectWindow** if the window is not the active window. Otherwise, handle the event according to your application.
wInGoAway	Button pressed in active window's close region. Call **TrackGoAway**. If **TrackClose** returns TRUE, call **CloseWindow**, or **HideWindow**, perhaps after saving whatever the user was working on inside the window.
wInZoom	Button pressed in active window's zoom region. Call **TrackZoom**. If **TrackZoom** returns TRUE, call **ZoomWindow**.
wInGrow	Button pressed in active window's grow region. Call **GrowWindow**.

USING TASKMASTER

TaskMaster is a procedure that can handle many standard functions. **TaskMaster** is called instead of **GetNextEvent** and the first thing **TaskMaster** does, is call **GetNextEvent**. If there isn't an event ready, **TaskMaster** will return zero. If an event is ready, **TaskMaster** will look at the event and try to handle it. If the event can not be handled at all, the event code is returned and the application can handle the event just like returning from **GetNextEvent**. If **TaskMaster** can handle the event, it will call standard functions to try and complete the task. For example, if the user presses the mouse button in an active window's zoom region, **TaskMaster** will detect it and call **TrackZoom**, then call **ZoomWindow** if the user actually selects the zoom region, and return no event. However, sometimes **TaskMaster** can handle an event only up to a point. If the user presses the mouse in the active window's content region, **TaskMaster** will detect it, but won't be able to go any further, so it returns wInContent, which tells the application the mouse button is down the active window's content region.

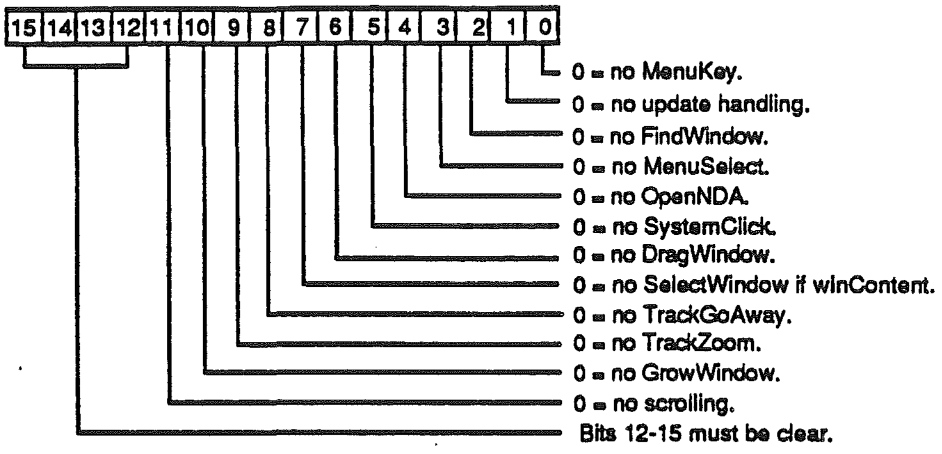
TaskMaster is provided for two reasons. The first is to make it easier for a programmer to get an application up and running as quickly as possible, and still take advantage of all the standard user interfaces. Although **TaskMaster** was tailored for beginning programmers, advanced programmers may find it useful when writing small, inhouse, applications. The second reason for **TaskMaster** is aimmed at applications that are sold and will hopefully be around for years, if not weeks, to come. In the future, applications might have to be modified to take advantage of some new, as yet unknown, feature. If an application is using **TaskMaster**, it may be possible to make the modification to **TaskMaster**, without adversely affecting past applications, so your application will be using the new feature without any modification on your part.

TaskMaster is one of the steps taken to remove the user interface duties from the application, as most operating system have done in the past. **TaskMaster** should be usable by even the most advanced applications, although some alternate algorithms may have to be used in order to get the desired results.

When calling **TaskMaster** you pass a pointer to a TaskMaster record, TaskRec. The beginning of the record is the same as an event record. When **TaskMaster** calls **GetNextEvent**, it will pass the pointer given, so the event record part of TaskRec is set by **GetNextEvent**. The structure of TaskRec is:

what	WORD	Event record portion, unchanged from GetNextEvent.
message	LONG	
when	LONG	
where	LONG	
modifiers	WORD	
TaskData	LONG	Extended portion for TaskMaster.
TaskMask	WORD	

The TaskMask is used by your application to tell **TaskMaster** about functions you would not like it to perform. To perform everything **TaskMaster** is capable of TaskMask should be $0FFF. Bits are clear in TaskMask to disable features. See **TaskMaster** function call for an outline of **TaskMaster** features and places TaskMask would cause a break. TaskMask is defined as:



It is important that bits 12-15 be clear. In fact, **TaskMaster** will return an error if they are not. The bits are for future features which will continue to run with predated applications because bits 12-15 will mask off the new, unknown to current applications, features.

Window type return codes from **TaskMaster** are:

inUpdate - The window, who's pointer is stored in the TaskData field of TaskRec, needs to be redrawn. Call **BeginUpdate**, draw the visRgn or the entire content region, and call **EndUpdate**.

wInMenuBar - The user made a selection from the system menu bar and the ID of the item selected is stored in the low-order WORD TaskData, the menu ID is in high-order WORD of TaskData. Handle the menu selection and unhighlight the menu's title when you have completed the requested action.

wInContent - This means the mouse button has been pressed inside the content region of the active window. The window's pointer is in the TaskData field of TaskRec. Because the event happen inside an area your application controls, it is up to you to handle this event in the manner you chose.

wInGoAway - User has selected the window's go-away region. Do what's needed and call **CloseWindow** with the window pointer in the TaskData field of TaskRec.

WINDOW MANAGER ICON FONT

The standard document window definition uses a font to draw the close and zoom boxes, and their highlighted states, in a window's title. If you would like to use different icons you can replace the default font. To replace the icon font, or just get the handle to the current font, call **SetWMgrIcons**. The format of the font is as follows:

Character 0	Close box.
Character 1	Highlighted close and zoom boxes.
Character 2	Zoom box.

WINDOW RECORDS

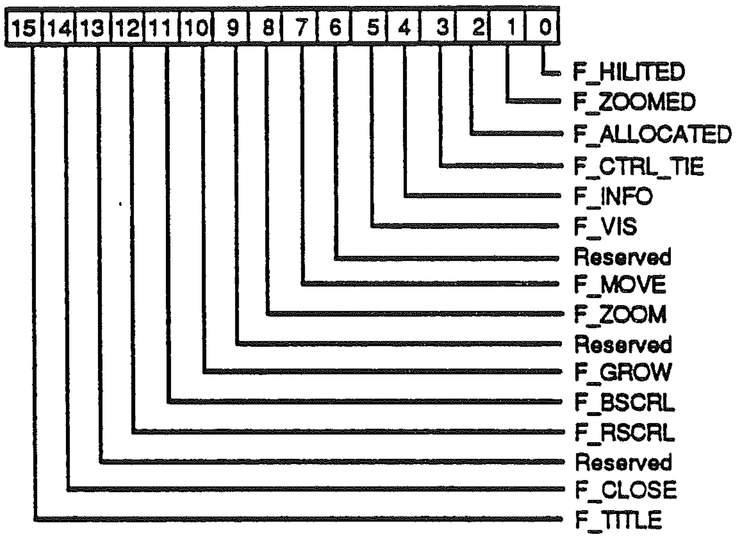
The Window Manager keeps all the information it requires for its operations on a particular window in a **window record**. The record contains the window's grafPort, title pointer, position, size of work area, a reserved long for the application, and other flags the Window Manager needs to manage windows. The complete window record is accessed directly only by the Window Manager. Application access to record information is restricted to calls through the Window Manager and directly to the first part of the window record.

Not allowing direct access to the entire window record has good and bad sides. Access to window information will be slower if calls to the Window Manager have to be made. However, the delay would have to be measured in miliseconds, and the delay never seen on the screen. On the plus side, future Window Managers would not be tied to an older, possibly inadequate, record structure. The chances of improving the current Window Manager, and maintaining compatibility accross future hardware, is greatly improved by allowing records to change.

Many Window Manager calls require a window pointer that is returned from **NewWindow**. That pointer is the pointer to the window's grafPort.

The part of the window record that is defined is:

wnext	LONG	Pointer to the next window in the window list.
wport	BYTE[186]	Window's port. Returned window pointers return a pointer to here.
wstrucRgn	LONG	Handle of window's entire region, the frame plus content.
wcontRgn	LONG	Handle of window's content region.
wupdateRgn	LONG	Handle of region that is the part of the content that needs redrawing.
wcontrol	LONG	Handle of application's first control in content region.
wFrameCtrl	LONG	Handle of frame's first control.
wframe	WORD	Bit vector that describes window.



F_HILITED	1 = frame is highlighted, 0 = unhighlighted.
F_ZOOMED	1 = currently zoomed, 0 = not zoomed.
F_ALLOCATED	1 = record was allocated, 0 = record was provided by application.
F_CTRL_TIE	1 = control's state is independent, 0 = inactive window has inactive controls.
F_INFO	1 = information bar, 0 = no information bar.
F_VIS	1 = currently visible, 0 = window is invisible.
F_MOVE	1 = title bar is a drag region, 0 = no drag region.
F_ZOOM	1 = zoom box on title bar, 0 = no zoom box. (Zoom box must have title bar.)
F_GROW	1 = grow box, 0 = no grow box. (Grow box must have at least one scroll bar.)
F_BSCRL	1 = window frame horizontal scroll bar, 0 = no horizontal scroll bar.
F_RSCRL	1 = window frame vertical scroll bar, 0 = no vertical scroll bar.
F_CLOSE	1 = close box, 0 = no close box. (Close box must have title bar.)
F_TITLE	1 = title bar, 0 = no title bar.

(The remainder of the record is undefined)

WINDOWS AND GRAFPORTS

It's easy for applications to use windows: To the application, a window is a grafPort that it can draw into with QuickDraw routines. When you create a window, you specify a rectangle that becomes the portRect of the grafPort in which the window contents will be drawn. The bit map for this grafPort, its pen pattern, and other characteristics are the same as the default values set by QuickDraw. These characteristics will apply whenever the application draws in the window, and they can easily be changed with QuickDraw routines.

There is, however, more to a window than just the grafPort that the application draws in. The other part of a window is called the **window frame**, since it usually surrounds the rest of the window. For drawing window frames, the Window Manager creates a grafPort that has the entire screen as its portRect; this grafPort is called the **Window Manager port.**

WINDOW FRAME COLORS AND PATTERNS

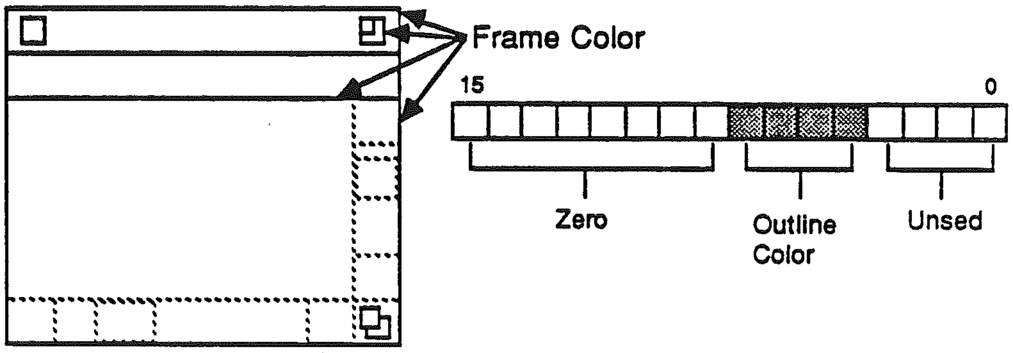
In addition the to the standard window types and controls, the color of the window and controls can be selected. Colors are selected from a color table which you either pass when creating a window, or a default table. The color table for a document window is:

FrameColor	WORD	Color of window frame.
TitleColor	WORD	Color of inactive bar, inactive title, and active title.
TBarColor	WORD	Color and pattern of active title bar.
GrowColor	WORD	Color of grow box.
InfoColor	WORD	Color information bar background.

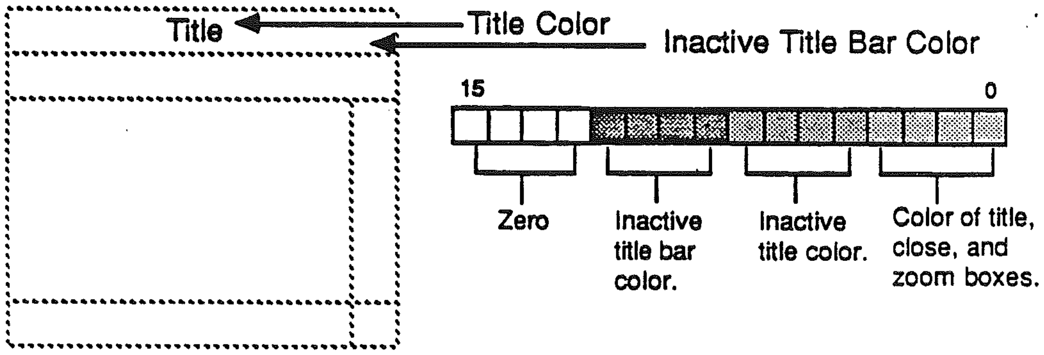
Use **SetFrameColor** to set the color table a window should use, and **GetFrameColor** to get a pointer to the window's current color table.

The following diagrams show how these colors are used.

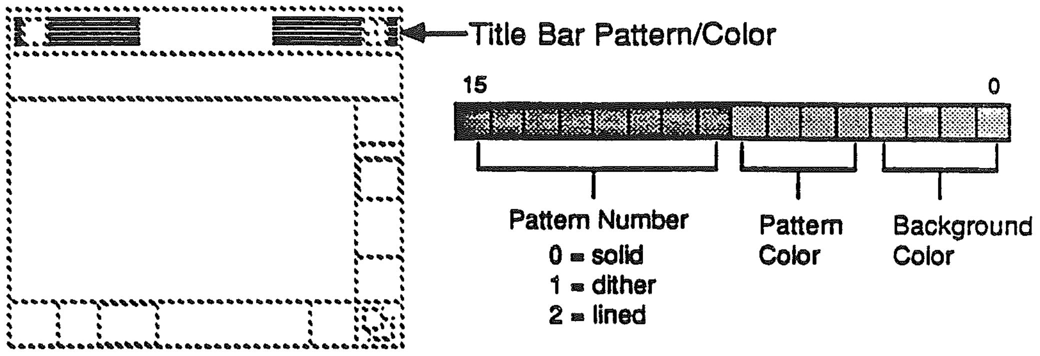
FrameColor:



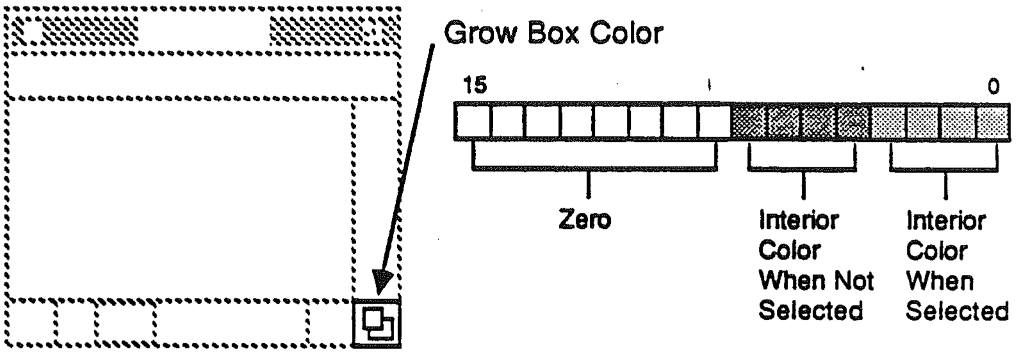
TitleColor:



TBarColor:

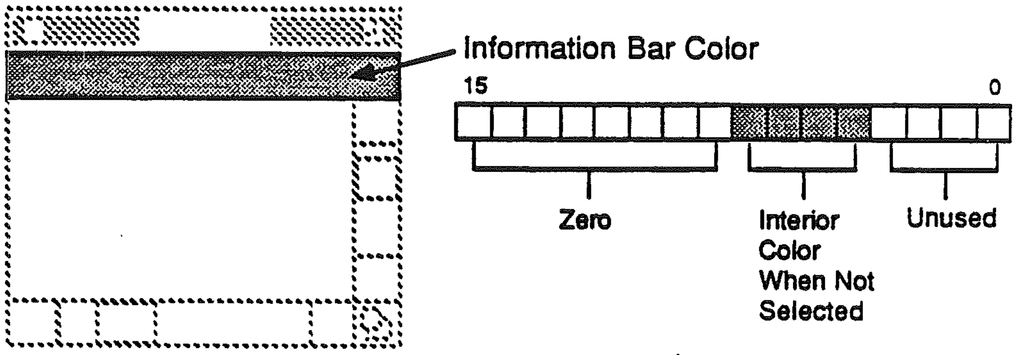


GrowColor:



Grow Box Color
15
0
Zero
Interior Color When Not Selected
Interior Color When Selected

InfoColor:



HOW A WINDOW IS DRAWN

When a window is drawn or redrawn, the following two-step process usually takes places: the window frame is drawn, then the window contents.

To perform the first step of this process, the Window Manager manipulates regions of the Window Manager port as necessary to ensure that only what should be drawn is drawn. It then calls the window definition function with a request that the window frame be drawn. The window definition function is either within the Window Manager or in the application for custom windows (see DEFINING YOUR OWN WINDOWS).

For the second step the Window Manager generates an **update event** to get the application to draw the window contents. It does this by accumulating, in the update region, the areas of the window's content region that need updating. The Event Manager periodically calls **CheckUpdate** to see if there's any window whose update region is not empty; if it finds one, it reports (via the **GetNextEvent** function) that an update event has occurred, and passes along the window pointer in the event message. Update events will be issued to the front most window first and the bottom most last. The application should respond as follows:

1. Call **BeginUpdate**. This procedure temporarily replaces the visRgn of the window's grafPort with the intersection of the visRgn and the update region. It then clears the update region for that window.

2. Draw the window contents.

3. Call **EndUpdate** to restore the actual visRgn.

DRAW CONTENT ROUTINE

If wContDefProc is nonzero, the value will be considered the address of a routine in your application that will draw the window's content region. WContDefProc must be set if you want to use window frame scroll bars. **TaskMaster** will scroll the content and call wContDefProc to update the uncovered area when the use perfroms a scrolling action. WContDefProc could be considered a control action procedure.

WContDefProc might be useful even if you are not using window frame scroll bars. **TaskMaster** will be able to handle you update events if wContDefProc is set. **TaskMaster** will call **BeginUpdate**, wContDefProc and **EndUpdate**. (**TaskMaster** does take some short cuts in calling **BeginUpdate** and **EndUpdate** because its part of the Window Manager.) Along these lines, if you are using window frame scroll bars, and therefore **TaskMaster**, you will not get update events, because they are handled by **TaskMaster**.

There are no inputs or outputs to your draw content routine. Simple draw what is needed in the content and perform a RTL to exit. Remember that the content will have already been erased using the window port's background pattern, and the visRgn is set to the area needing to be redrawn.

> **Warning:** Do not change ports or perform a **SetOrigin** call while in your draw content routine.

DRAW INFORMATION BAR ROUTINE

If bit 4 in wFrame of the **NewWindow** parameter list is set, the window will have an information bar. An information bar is the width of window and a bit higher than the system font. It will appear just above the content region.

The Window Manager draws the information bar, but it is up to the application to draw any information inside. Your application can do this by storing the address of an information bar draw routine in wInfoDefProc of the **NewWindow** parameter list (also set bit 4 of wFrame). When the standard window frame defProc draws the empty information bar, it will also call wInfoDefProc. The inputs to your routine will be:

InfoBar:LONG	Pointer to information bar enclosing RECT.
InfoData:LONG	wInfoRefCon value from **NewWindow** parameter list.
theWindow:LONG	Pointer to window's port.

An information bar draw routine that just prints a string might look like this:

```
InfoDefProc  START
;
theWindow    equ   6
InfoData     equ   theWindow+4
InfoBar      equ   InfoData+4
;
;
             phd                            Save the current direct page.
;
             tsc
             tcd                            Switch to direct page in stack.
;
;
; --- Position the pen at the text starting point ---------------------------
;
             ldy   #left_side               (Where left_side equals 2.)
             lda   [InfoBar],y              Get the left side of the information bar,
             clc
             adc   #20                      plus a tab over,
             pha                            to get a starting x position.  (Pass to _MoveTo.)
;
             ldy   #top_side                (Where top_side equals 0.)
             lda   [InfoBar],y              Get the top side of the information bar,
             clc
             adc   #10                      plus enough to vertically center the text,
             pha                            to get a starting y position.  (Pass to _MoveTo.)
;
             _MoveTo                        Move the pen to the starting point.
;
;
; --- Print the text on the information bar ---------------------------------
;
             pea   infoStrg|-16             Pass high word of string.
             pea   infoStrg                 Pass low word of string.
             _DrawString                    Print the string.
;
```

```
;
; — All done, now clean-up stack and return to Window Manager ————————
;
            ply                      Get original direct page back.
;
            lda     2,s              Move return down over input parameters.
            sta     <14              Works because stack and direct page are equal.
            lda     0,s
            sta     <12
;
            tsc                      Now move stack pointer over input parameters.
            clc
            adc     #12              Number of bytes of input parameters.
            tcs                      New stack.
;
            tya                      Restore original direct page.
            tcd
;
            rtl                      Back to Window Manager.
;
            END
```

I have taken some liberties here, such as taking for granted the color and writing mode of the pen when the text is written. When entered, the current port is the Window Manager's. It is permissible to change the pen location, color, and writing mode without saving the original port state. However, that's as much as you should do without first saving the port state, and then restoring it on exit.

Another liberty taken is when the text is centered vertically. You should make QuickDraw calls to find font height, find the InfoBar height, and then actually center the text. You should always use InfoBar as offsets into the information bar interior, they could be different from time to time.

And of course 'infoStrg' would have to be defined.

MAKING A WINDOW ACTIVE: ACTIVATE EVENTS

A number of Window Manager routines change the state of a window from inactive to active or from active to inactive. For each such change, the Window Manager generates an **activate event**, passing along the window pointer in the event message. The activeFlag bit in the modifiers field of the event record is set if the window has become active, or cleared if it has become inactive.

When the Event Manager finds out from the Window Manager that an activate event has been generated, it passes the event on to the application (via the GetNextEvent function). Activate events have the highest priority of any type of event.

Usually when one window becomes active another becomes inactive, and vice versa, so activate events are most commonly generated in pairs. When this happens, the Window Manager generates first the event for the window becoming inactive, and then the event for the window becoming active. Sometimes only a single activate event is generatred, such as when there's only one window in the window list, or when the active window is permanently disposed of (since it no longer exists).

Activate events for dialog and alert windows are handled by the Dialog Manager. In response to activate or inactivate events for windows created directly by your application, you might take actions such as the following:

- Inactivate controls in inactive window, and activate controls in active windows.
- In a window that contains text being edited, remove the highlighting or blinking cursor from the text when the window becomes inactive and restore it when the window becomes active.
- Enable or disable a menu or certain menu items as appropriate to match what the user can do when windows become active or inactive.

DEFINING YOUR OWN WINDOWS

You may want to define your own type of window - maybe a round or hexagonal window, or even a window shaped like an apple. QuickDraw and the Window Manager make it possible for you to do this.

To define your own type of window, you write a routine that can will dupicate some Window Manager functions. When the Window Manager needs to do something it will call your routine and not its own. The address of the routine is passed to CreateWindow. The inputs to your routine will be:

```
            varCode:WORD - operation needed to be performed.
            theWindow:LONG - pointer to the window's port.
            Param:LONG - flag used by some messages.
```

Output will be:

```
            outCome:LONG - returned flag.
```

Offsets into the stack are:

Param	= 4
theWindow	= Param+4
varCode	= theWindow+4
outCome	= varCode+2

Your routine must strip off the three input parameters and return via RTL. So, the shell of your defProc routine might be:

```
MyWindow  START
          lda   12,s                 Get varCode.
          asl   a
          tax
          lda   >actions,x
          pha
          rts                        Go to action handler.

actions   dc    i2'draw_wind-1'      Routine to draw the window's frame.
          dc    i2'test_hit-1'       Routine that find a window region at a given point.
          dc    i2'calc_rgns-1'      Compute the window's structRgn and contRgn.
          dc    i2'init_wind-1'      Do additional initialization.
          dc    i2'kill_wind-1'      Do additional disposal.
          END

draw_wind START
          :
      Draw window frame.
          :
          jmp  exit
          END

test_hit  START
          :
      Find what area of the window the point in Param is located.
          :
          jmp  exit
          END

calc_rgns START
          :
      Compute the window's structRgn and contRgn.
          :
          jmp  exit
          END

init_wind START
          :
      Perform additional initialization.
          :
          jmp  exit
          END

kill_wind START
          :
      Perform additional disposal.
          :
          jmp  exit
          END
```

```
exit            START
;
                lda   2,s                    Move return address.
                sta   12,s
                lda   1,s
                sta   11,s
;
                tsc                          Strip off input parameters.
                sec
                sbc   #10
                tcs
;
                rtl                          Return to Window Manager.
                END
```

varCode will be:

wDraw	= 0	Draw window frame.
wHit	= 1	Tell what region mouse button was pressed in.
wCalcRgns	= 2	Calculate wstrucRgn and wcontRgn.
wNew	= 3	Do any additional window initialization.
wDispose	= 4	Take any additional disposal actions.

The following sections tell you what is expected is response to the varCode.

wDraw - Draw Window Frame

Param:

wDrawFrame	= 0	Draw the window's entire frame.
wInGoAway	= 1	Draw go-away region.
wInZoom	= 2	Draw zoom region.
Bit 31	= 1	Highlight.
	= 0	Unhighlighted.

Your routine should draw in the current grafPort, which will be the Window Manager port. The Window Manager will request this operation only if the window is visible.

Param	
$00000000	The entire window frame should be drawn as an inactive window.
$80000000	The entire window frame should be drawn as an active window.
$00000001	The go-away region should be drawn as unhighlighted.
$80000001	The go-away region should be drawn as highlighted.
$00000002	The zoom region should be drawn as unhighlighted.
$80000002	The zoom region should be drawn as highlighted.

wHit - Find What Region a Point Is In

Param equals the point to check. The vertical coordinate is in the low-order WORD and the horizontal coordinate in the high-order WORD. The Window Manager will request this operation only if the window is visible. Your routine should determine where the point is in your window and then return:

wNoHit	= 0	Not on the window at all.
wInContent	= 19	In window's content region.
wInDrag	= 20	In window's drag (title bar) region.
wInGrow	= 21	In window's grow (size box) region.
wInGoAway	= 22	In window's go-away (close box) region.
wInZoom	= 23	In window's zoom (zoom box) region.
wInInfo	= 24	In window's information bar.
wInFrame	= 27	In window, but not any of the above areas.

Usually, wNoHit means the given point isn't anywhere within the window, but this is not necessarily so.

wCalcRgns - Calculate Window's Regions

Your routine should calculate the window's entire region and its content region based on the current grafPort's portRect. The Window Manager will request this operation only if the window is visible. When you calculate regions for your window, do not alter the clipRgn or visRgn of the window's grafPort. The Window Manager and QuickDraw take care of this for you. Altering the clipRgn or visRgn may result in damage to other windows.

wNew - Initialization

After initializing fields as appropriate when creating a new window, the Window Manager sends the message wNew to your routine. This gives your routine a chance to perform any initialization it may require. For example, because the structure of the window record is not documented you made want to allocate your own record structure, initialize it, and store its pointer via SetWRefCon.

wDispose - Remove Window

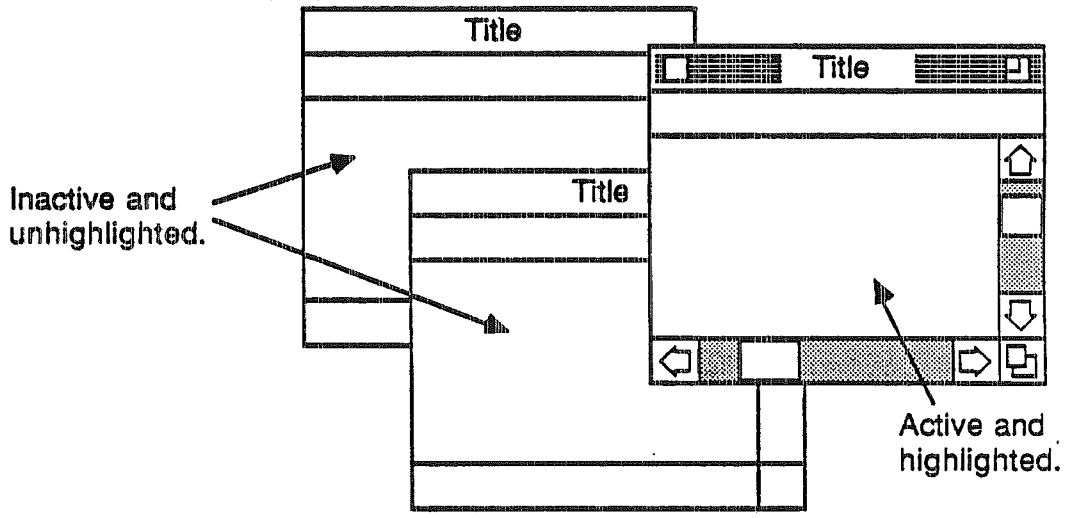
The Window Manager's CloseWindow and DisposeWindow procedures send this message so your routine can carry out any additional actions required when disposing of the window. The routine might, for example, release space that was allocated by the initialize routine.

wGrow - Draw the Outline of the Window

Param is a pointer to a RECT (rectangle). Your routine should draw an outline image of your window that would fit the given rectangle. The Window Manager requests this operation repeatedly as the user drags inside the grow region. Your routine should use the grafPort's current pen pattern and pen mode, which are set up so one call will draw the outline and next will erase it (XOR mode).

Definitions

highlighted	The frame of the window is drawn in full detail. Generally the frontmost window is the only highlighted window on the screen. However, any window could be highlighted, even all the windows.
unhighlighted	Opposite of highlighted. Generally all the windows behind the frontmost window are unhighlighted. However, even the frontmost window can be unhighlighted.
active	In this document it only means the frontmost window on the screen. But generally it also means it is highlighted. This should also be the window your application acts on when the user types, gives commands, or whatever is appropriate to the application.
inactive	Any windows behind the frontmost (active) window. Generally these window will be unhighlighted.



window list or list	An internal linked list of all the window records created by NewWindow and not removed by a CloseWindow or DisposeWindow call. The first visible window in this list is the active window.
top window or top	The first window in the window list. However, the top window is not the active window unless it is visible.
bottom window or bottom	The last window in the window list.