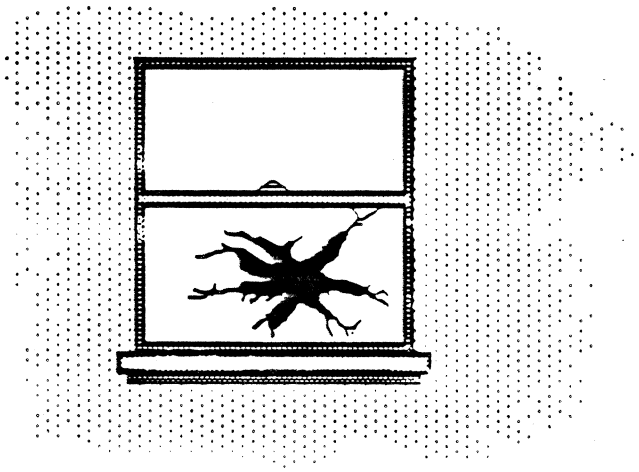
Cortland Window Manager

Dan Oliver

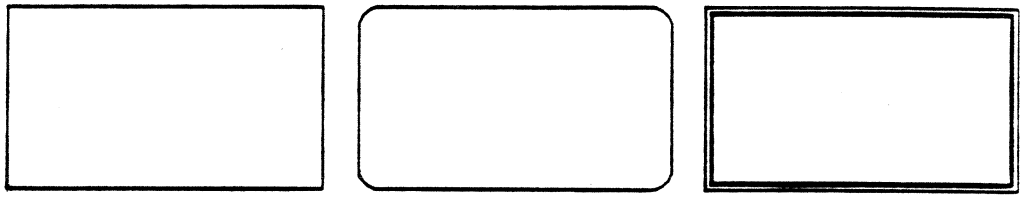


Initial release 01/30/86

ABOUT THE WINDOW MANAGER

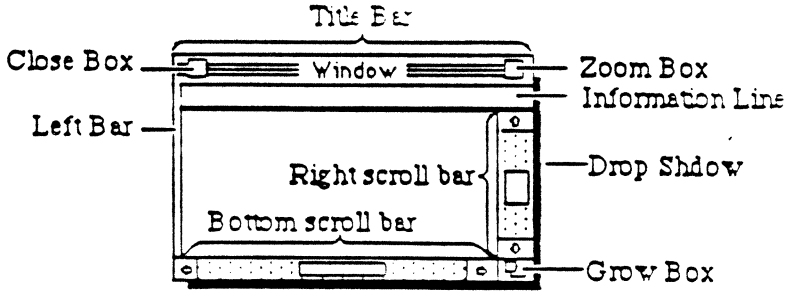
The Window Manager is a tool for dealing with windows on the Cortland screen. The screen represents a working surface or **desktop**; graphic objects appear on the desktop and can be manipulated with a mouse. A **window** is an object on the desktop that presents information, suc as a document or a message.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

Some standard types of windows are predefined; **square cornered, rounded corner**, and **alert**.



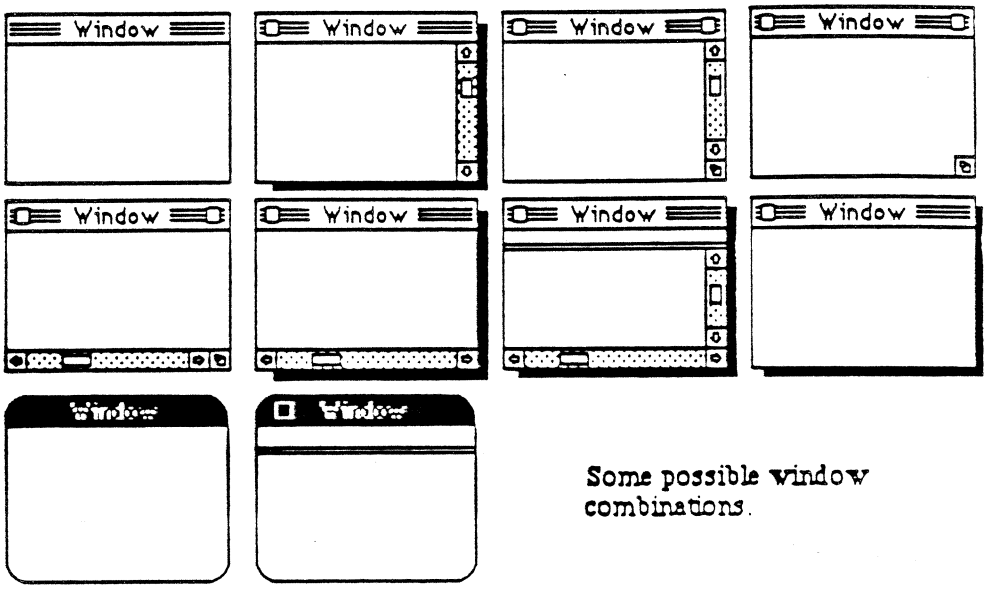
Square Cornered Round Cornered Alert

Inside the square and round cornered windows can be **standard window controls**, which are; title bar, close box, zoom box, right scroll bar, bottom scroll bar, grow box, information bar, left bar, and drop shadow. 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 i maximum size and then to return it to its previous size and position. The right scroll bar is used t scroll vertically through the data in the window. The bottom scroll bar is used to scroll horizontal 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 b effected by the scroll bars. The left bar is a thickened left side of a window which can help separ data in different windows when they overlap. The drop shadow helps separates windows from one another and can also be used as a move region for moving the window off the top of the desktop.



A square window may have any or all of the standard window controls. The only restriction is th 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.

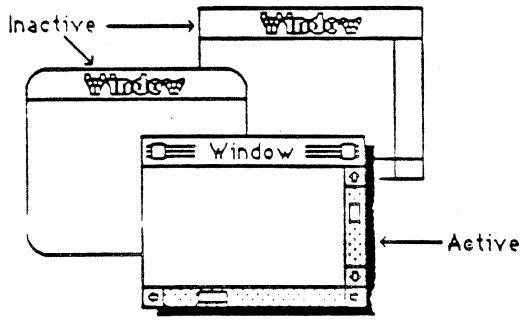
The only standard controls that may be added to a round cornered window is a title bar, close box, information bar, and drop shadow. And no standard controls may be added to a alert window.



Some possible window combinations.

Your application can easily create standard window types, or define your own window types (see Create 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 box. 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.

A window that is frontmost is called the **active** window. It appears **highlighted** - displayed in a distinctive way, so it stands out from other windows. There can only be one active window on the screen at any one time. This will be the one that will be acted on when the user types, gives commands, or whatever is appropriate to the application being used. Any other windows are **inactive**, appear behind the active window and are not highlighted.



The Window Manager's main function is to keep track of overlapping windows. You can draw i 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 an newly exposed areas and provides a convenient machanism for you to ensure that they are proper 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 representati 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 butto 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 FRAME COLORS AND PATTERNS

In addition the to the standard window types and controls, the color of the window and controls can be selected. The data type used to set the pattern and color is:

Where the colors are indexes into the current color table. Changing what the color is for the current color table is not done here. Pattern number is one of a standard pattern type generated by the Window Manager. The standard pattern table is:

0 Solid, this is the default for close box interior, content region, and thumbs.
1 Dither.
2 Dotted, default for scroll bars.
3 Lined, default for title bar.
4 Zoom, default for zoom box.
5 Sizer, default for grow box.

Hopefully this list will grow.

The patterns/colors, for each part of the frame, are set by a call to SetFrame with a pointer to this 23 word table:

0	Frame Outline
1	Frame Interior
2	Content Interior
3	Drop Shadow
4	Title Bar Interior
5	Title (color)
6	Close Box Interior
7	Close Box Outline
8	Zoom Box Interior
9	Zoom Box Outline
10	Information Bar Interior
11	Up Arrow Interior

12	Right Scroll Bar Interior
13	Right Thumb Interior
14	Down Arrow Interior
15	Right Scroll Bar Outline
16	Grow Box Interior
17	Grow Box Outline
18	Right Arrow Interior
19	Bottom Scroll Bar Interior
20	Bottom Thumb Interior
21	Left Arrow Interior
22	Bottom Scroll Bar Outline

GetFrame is used to get the current settings of a window.

WINDOWS AND GRAFPORTS

It's easy for applications to use windows: To the application, a window is a grafPort that it can draw into like any other 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, except for the character font, which is set to the application font. These characteristics will apply whenever the application draws in the window, and they can easily be changed with QuickDraw routines (SetPort to make the grafPort the current port, and other routines as appropriate).

There is, however, more to a window than just the grafPort that the application draws in. In a standard window any standard controls are drawn by the Window Manager, not by the application.

The part of a window the Window Manager draws 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 REGIONS

Every window has the following two regions:

- The **content region**: the area that your application draw in

- The **structure region**: the entire window; frame plus content.

The content region is bounded by the rectangle you specify when you create the window (that is, the portRect of the window's grafPort) This 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. Clicking in this region toggles between the current position and size to a maximum size, and back again.

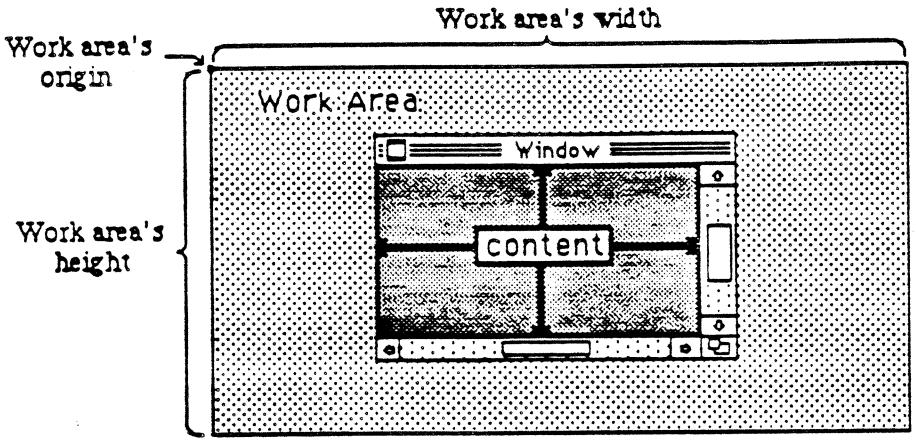
Clicking in any region of an inactive window simply makes it the active window.

> Note: The results of clicking and dragging that are discussed here don't happen automatically; 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 hardware (screen) can display at one time. The name window is used because the user sees through the window onto a larger area. The power of windows is their ability to give the user a standard device for scrolling through a large amount 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 work area is what the microfiche is to the viewer. Through the content region the

user can see part of the **work area**, unless the content region is large enough to view the entire work area. Scroll bars are used to view different parts of the work area. The grow box and zoom box are used to display more or less of the work area at one time. When the window is moved the work area is moved with it, so the view in the content would remain the same.

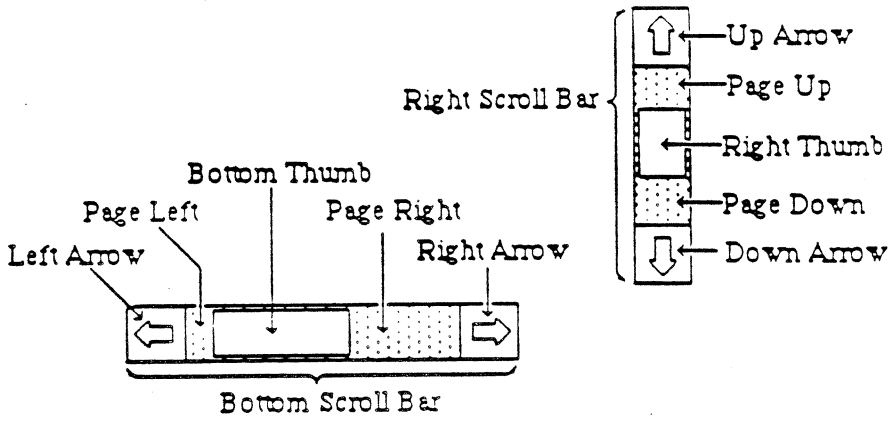


The origin, width, and height of the work area is set when the window is created, and changed if needed by calling the Window Manager.

WINDOW SCROLL BARS

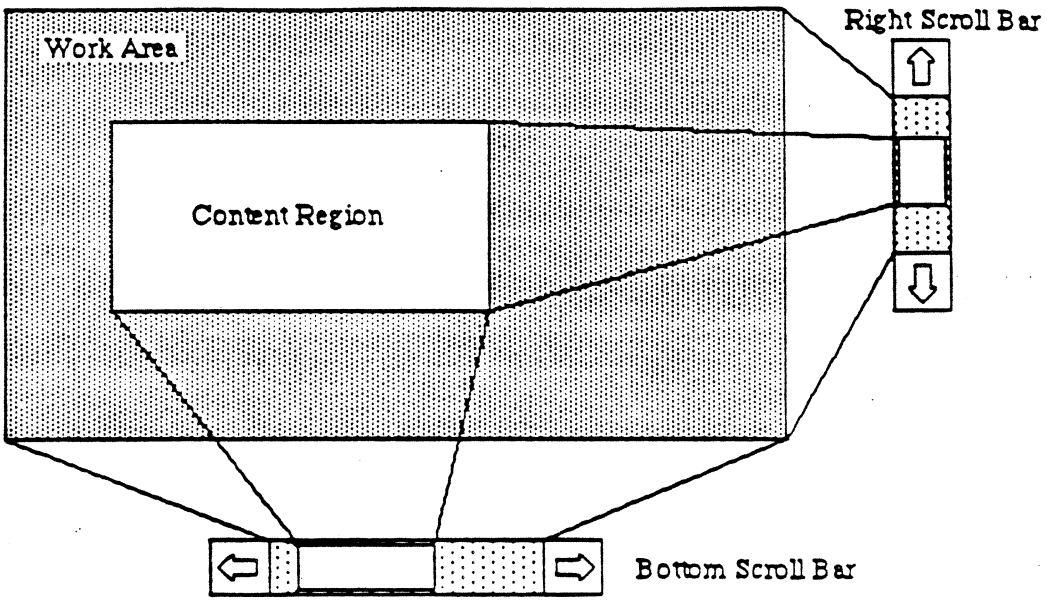
Window scroll bars are the devices used for scrolling the work area through the content region and showing the relationship between the work area and content region. The Control Manager must be installed in order to use scroll bars in windows. Scroll bars are defined by the Control Manager but this document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

For clarity we will expand the names given to the parts of the scroll bar by the Control Manager.



Scroll bars also show information about the work area. The size of the scroll thumb gives the ratio

of what is seen in the content region and what isn't and its position on the scroll bar shows the exact location being viewed. The scroll bar is like a shrunken cross section of the work area.



WINDOW RECORDS

The Window Manager keeps all the information it requires for its operations on a particular windo 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 the screen.

(A detailed report about the record will be added when the Window Manager is further along.)

HOW A WINDOW IS DRAWN

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

To perform the first step of this process, the Window Manager manipulates regions of the Windo 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 redrawn. The window definition function is either within the Window Manager or in the application for custom window (see Create Your Own Windows).

Usually the second step is that 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 Toolbox Event Manager periodically checks 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 flag for that window.

2. Draw the window contents.

3. Call EndUpdate, which restores the actual visRgn.

The Window Manager allows an alternative to the update event machanism that may be useful for applications that want to use it. A pointer to a routine in the application that would handle the drawing of the content can be passed when a window is created. The Window Manager would handle the BeginUpdate and EndUpdate and will call the routine directly saving alittle time and your application would never have to worry about BeginUpdate, EndUpdate, and checking the event for update messages.

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 Toolbox 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 generated, 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:

- In a window that contains controls like scroll bars, other than those in the window frame, erase the inside of a control to show it can not be used on an inactivate event. The controls can then be redrawn in full on an activate event.

- 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 the window becomes active or inactive.

USING THE WINDOW MANAGER

To use the Window Manager, you must have previously called InitGraf to initialize QuickDraw. The first Window Manager routine to call is the initialization routine InitWindows with a page in bank zero it can use as its zero page (the Event Manager and Menu Manager need to have the same page number).

Where appropriate in your program, use NewWindow to create any windows you need. You can supply a pointer to the storage for the window record or let it be allocated by the Window Manager. When the window is no longer needed call CloseWindow if you supplied the storage, or DisposeWindow if not.

Then you just wait for an event by calling GetNextEvent and handle the following events in the following ways:

- For an update event call BeginUpdate, draw the visRgn or the entire content region, and call EndUpdate.

- For a mouse-down event, call the FindWindow function to find out which part of which window the mouse button was pressed in.

    - If it was pressed in the content region of an inactive window, make that window the active window by calling SelectWindow.

    - If it was pressed in the grow region of the active window, call GrowWindow to pull around an image that shows how the window's size will change, and then SizeWindow to actually change the size.

    - If it was pressed in the drag region of any window, call DragWindow, which will pull an outline of the window across the screen, move the window to a new location, and, if the window is inactive, make it the active window (unless the Command key was held down).

    - If it was pressed in the go-away region of the active window, call TrackGoAway to handle the highlighting of the go-away region and to determine whether the mouse is inside the region when the button is released. Then do whatever is appropriate as a response to this mouse action in the particular application. For example, call CloseWindow or DisposeWindow if you want the window to go away permanently, or HideWindow if you want it to disappear temporarily.

The MoveWindow procedure simply moves a window without pulling around an outline of it. Note, however, that the application shouldn't surprise the user by moving (or sizing) windows unexpectedly. There are other routines that you normally won't need to use that let you change the title of a window, place one window behind another, make a window visible or invisible, and

access miscellaneous fields of the window record.

There is a rourine that may be of some use to applications that have standard window that act in standard ways. The routine is called TaskMaster and is called with a pointer to the event record right after a valid event has been received from the Event Manager. The TaskMaster will handle activate and inactivate messages, button down on inactive windows. The TaskMaster will also handle all button downs and drags on the active window frame and standard window controls. If the TaskMaster handles the event it will return a null event and your application can return to pollin GetNextEvent. TaskMaster will return handle event if the event didn't effect any Window Manager functions, if the button was down in the content region, or the close box was selected. By using TaskMaster your application would only need to call InitWindows, NewWindow and CloseWindow in order to have full function windows with user interaction.

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:

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

Output will be: outCome:LONG - returned flag depending on message outcome.

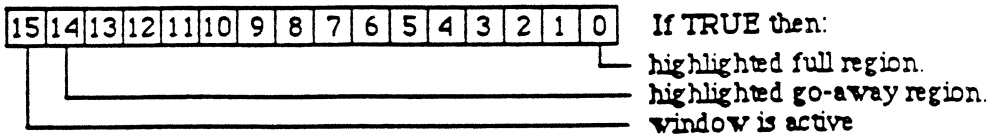
message will be:

wDraw	= 0	Draw window frame.
wHit	= 1	What region is at the point passed.
wCalcRgns	= 2	Calculate window and content regions.
wNew	= 3	Initialization.
wDispose	= 4	Take any pisposal actions.
wGrow	= 5	Draw outline of window.
wDrawGIcon	= 6	Draw size box in content region.

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

wDraw - Draw Window Frame

Param:



Your routine should draw the window frame in the current grafPort, which will be the Window Manager port. This routine should make certain checks to determine exactly what it should do. If the visible field in the window record is FALSE, the routine should do nothing. Otherwise, it should draw the entire window frame. If the window active flag is TRUE in Param the frame should be drawn highlighted in whatever way is appropriate to show that this is the active window. If the highlighted flag in Param is TRUE for either zoom or go-away, the window frame should be highlighted in whatever way is appropriate to show it.

wHit - Find What Region a Point Is In

Param equals the point to check. The vertical coordinate is in the high-order WORD and the horizontal coordinate in the low-order WORD. Your routine should determine where the point is i your window and then return:

wNoHit	= 0	None of the following.
wInContent	= 1	In the content region.
wInDrag	= 2	In drag region.
wInGrow	= 3	In grow region, active window only.
wInGoAway	= 4	In go-away region, active window only.

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 asn its content region based on the curre 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 may require. For example, if the content region is unusually shaped, the initialize routine might allocate space for the region and store the region handle in the application's reserved LONG in the window record.

wDispose - Remove Window

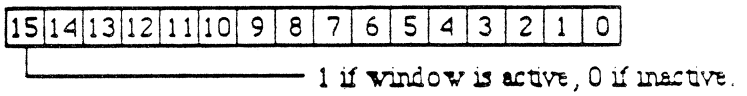
The Window Manager's CloseWindow and DisposeWindow procedures send this message so your routine can carry out any additional actions required when disposing of the window. The routine might, for example, release space that was allocated by the initialize routine.

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).

wDrawGIcon - Draw the Size Box

Param:



Your routine should draw the grow region in the window. The active flag in Param should be check to draw an inactive grow region if the window is inactive.

WINDOW MANAGER ROUTINES

Text in *Italic* refers to features not yet implemented.

Conventions:

Zero page must be in bank zero.
The Window Manager's zero page must always be on a page boundary.
A window record cannot straddle a memory bank.
A window record cannot start at $0000 in any bank.

These conventions are to cut code size but may be too restrictive, let's wait a bit and see.

Initialization and Termination

BootWmgr Call #1

input: None.
output: None.

Called only by **SetTSPtr**.

InitWindows Call #2

input: wZeroPage:WORD - page number Window Manager can use for zero page.
Zero page must be on page boundary.

output: None

Calls SetWAP to inform the Tool Locator which zero page number the Window Manage will use.
Clears the window list.
Sets the default desktop pattern and color.
Opens the Window Manager's port.
Draws the desktop which is the entire screen or the area below the system menu bar if the is a system menu bar.

TermWindows Call #3

input: None.
output: None.

Free any memory allocated by the Window manager.

NewWindow Call #5

inputs: wParams:LONG - pointer to initialization block (defined on the next page).
Memory will be allocated for the record if the pointer is zero.
wPlane:LONG - pointer to window that this window should appear behind or:
0 to not display window (nserted as top window in list).
-1 makes it the top most.
-2 makes it the bottom most.
wStorage:LONG - pointer to memory for window record.

output: theWindow:LONG - pointer to window record (in case it was allocated).
Zero if error.

Opens a port for the window.
Adds the window to the window list according to wPlane.
Displays the window if wPlane is not zero.

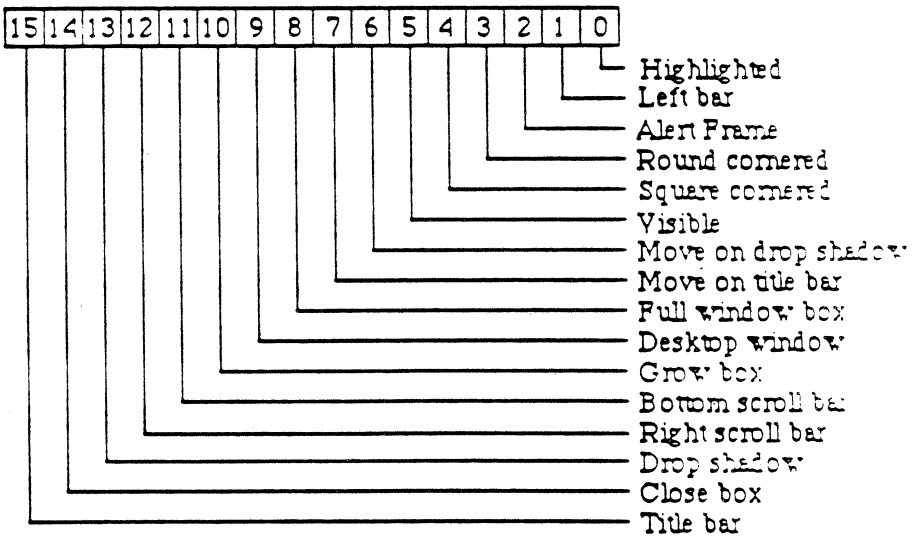
wParams - Initialization Block:

RECT - initial position and size of window:

Top
Left side
Bottom
Right side

If Top = $7FFF then the window manager will position the window and Bottom and Right side will be used as Width and Height.
If Bottom = $7FFF then the window manager will select a size for the window and Top, Left side will be used as the window's origin.
If both Top and Bottom = $7FFF then both the position and size will be selected by the window manager.

LONG - zero to get update events for redrawing the content, or the address of a routine that can be called directly by Window Manager that would redraw window's content.
POINT - origin of the work area.
WORD - width of the work area in pixels.
WORD - height of the work area in pixels.
WORD - window frame type:



LONG - pointer to window's title.
LONG - pointer to window's control list (not standard window controls).

CloseWindow Call #7

input: theWindow:LONG - pointer to window's record.
output: None.

theWindow must be in the window list.
Removes theWindow from the screen.
Removes theWindow from the window list.
Called to remove a window that was created by **NewWindow** using a non-zero wStorage pointer.

DisposeWindow Call #8

input: theWindow:LONG - pointer to window's record.
output: None.

theWindow must be in the window list.
Calls CloseWindow
Frees memory used by window's record.
Called to remove a window that was created by **NewWindow** using a zero wStorage pointer.

Window Display

SetWTitle Call #9

inputs: title:LONG - pointer to string for new title.
theWindow:LONG - pointer to window's record.
output: None.

theWindow must be in the window list.
Updates window's record with new title.
Redraws window's frame.

GetWTitle Call #10

input: theWindow:LONG - pointer to window's record.
output: title:LONG - pointer to string of window's title.

SetFrame Call #11

inputs: newColor:LONG - pointer to 23 word pattern/color table.
theWindow:LONG - pointer to window's record.
output: None.

See WINDOW FRAME COLORS AND PATTERNS.

GetFrame Call #12

inputs: newColor:LONG - pointer to 23 word table that will be set with the color table.
theWindow:LONG - pointer to window's record.
output: None.

See WINDOW FRAME COLORS AND PATTERNS.

SelectWindow Call #13

input: theWindow:LONG - pointer to window's record.
output: None.

theWindow must be in the window list.
Inactivates current top window.
Activates theWindow.
Redraws the screen.

HideWindow Call #14

input: theWindow:LONG - pointer to window's record.
ouput: None.

theWindow must be in the window list.
Removes theWindow from the screen.
Makes next window activate if theWindow was the current active window.
Redraws the screen.
theWindow is taken out of the window list.

ShowWindow Call #15

inputs: showFlag:WORD - TRUE = show, FALSE = hide.
theWindow:LONG - pointer to window's record.
output: None.

theWindow must already be in the wndow list.
Draw theWindow on the screen under the top window if there is one.

SendBehind Call #16

inputs: behindWindow:LONG - pointer to window record or:
-2 to put theWindow behind all others.
theWindow:LONG - pointer to window's record.
output: None.

Both theWindow and behindWindow must be in the window list.
Window list is reordered.
Screen is redrawn.

FrontWindow Call #17

input: None.
output: theWindow:LONG - pointer to the active window's record.

HiliteWindow Call #30

input: hilite:WORD - TRUE to highlight window frame, FALSE to unhighlight.
theWindow:LONG - pointer to window's record.
output: None.

Sets or clears the highlight bit in the window's frame and redraws the frame. The bit will keep its state until another HiliteWindow or SelectWindow call.

ShowHide Call #31

input: visFlag:WORD - F_VIS ($0020) to show, zero to hide.
theWindow:LONG - pointer to the window's record.
ouput: None.

Never changes the highlighting or window order.

BringToFront Call #32

input: theWindow:LONG - pointer to window's record.
output: None.

Draws theWindow in front of all the others but does not highlight.

DrawGrowIcon Call #18 (not completed)

input: theWindow:LONG - pointer to window's record.
output: None.

theWindow must be in the window list.
Draws the grow box in the window.

User Interaction

FindWindow Call #19 (not completed)

inputs: whichWindow:LONG - address of where to store pointer of window.
thePT:POINT - x,y coordinate on screen to check.
outputs: Location:WORD:

0 = on desktop.	whichWindow = 0.
1 = on system menu bar.	whichWindow = 0.
2 = on system window.	whichWindow = window.
3 = on content region.	whichWindow = window.
4 = on drag region.	whichWindow = window.
5 = on grow box.	whichWindow = active window.
6 = on close box.	whichWindow = active window.
7 = on frame.	whichWindow = window.

On frame is a part of the window that isn't a move, grow, content, or close region.

TrackGoAway Call #20 (not completed)

inputs: startPt:POINT - starting point of cursor (where the button went down).
theWindow:LONG - pointer to window's record.
output: GoAway:WORD - TRUE = go away selected, else FALSE.

theWindow must be in the window list.
Watches cursor while button is held down.
Highlights close box when cursor is inside, normal when not inside.

MoveWindow Call #21 (not completed)

inputs: newPos:POINT - new origin of window.
theWindow:LONG - pointer to window's record.
output: None.

Window must be in window list.
Window is moved on the screen.

DragWindow Call #22 (not completed)

inputs: startPt:POINT - start point of cursor.
Bounds:RECT - cursor boundary.
theWindow:LONG - pointer to window's record.
output: None.

Window must be in window list.
Outline of window is drawn.
Cursor is tracked while the button is down.
Outline is moved:
(current cursor position bounded by Bounds) - startPt.
When cursor is released the outline is erased and the window moved via MoveWindow.

GrowWindow Call #23 (not completed)

inputs: minHeight:WORD - minimum height of window allowed.
minWidth:WORD - minimum width of window allowed.
maxHeight:WORD - maximum height of window allowed.
maxWidth:WORD - maximum width of window allowed.
startPT:POINT - starting point of cursor.
theWindow:LONG - pointer to window's record.
output: newSize:LONG - high WORD = new height, low WORD = new width.

Window must be in window list.
Outline of window is drawn.
Cursor is tracked while the button is down.
Lower right corner of outline is moved: current cursor position - startPt.
When cursor is released the outline is erased, the new size computed and returned.
See SizeWindow to resize the window.

SizeWindow Call #24 (not completed)

inputs: newWidth:WORD - new width of window.
newHeight:WORD - new height of window.
theWindow:LONG - pointer to window's record.
output: None.

Window must be in window list.
Window's size is changed while its origin remains the same.
Screen is redrawn.

TaskMaster Call #25 (not completed)

input: theEvent:LONG - pointer to an event record just returned from the Event Manager
output: Flag:WORD - FALSE for null event (may have been handled), TRUE if the event is valid and should be acted on.

See TASKMASTER.

Update Region

BeginUpdate Call #26 (not completed)

input: theWindow:LONG - pointer to window's record.
output: None.

theWindow must be in the window list.
Window's visRgn is replaced with the union of visRgn and update region.
Update region is then emptied.
The window's visRgn is then ready for updating.

EndUpdate Call #27 (not completed)

input: theWindow:LONG - pointer to window's record.
output: None.

theWindow must be in the window list.
Restore visRgn to full visible region.

Miscellaneous Routines

WmgrVersion Call #4

input: None.
output: wVersion:WORD - Window Manager's version number.

GetWMgrPort Call #28

input: None.
output: wPort:LONG - pointer to window manager's port.

PinRect Call #29 (not completed)

inputs: Bounds:RECT - boundary of given point.
thePt:POINT - any point.
output: newPt:POINT - point inside Bounds nearest to thePt.

CheckUpdate Call #6 (not completed)

input: theEvent:LONG - pointer to an even record.
output: Falg:WORD - TRUE if update event found, else FALSE.

This routine is called by the Event Manager. From the front to the back in the window list, it looks for a visible window that needs updating. If it finds one whose window record has a pointer to a redraw routine it calls it to complete the update and looks for the next visible window that needs updating. If it ever finds a window needing to be updated and whose window record doesn't contain a redraw routine, it stores an update event for that window in theEevnt and return TRUE. If it doesn't find such a window, it return FALSE.

SetWRefCon Call # (not completed, and in debate)

inputs: refCon:LONG - reserved LONG for application's use
theWindow:LONG - pointer to window's record.
output: None.

This call is used to set a LONG value that is inside the window record and is reservered for the application's use. This call, and calls like it, is in debate because it may be easier for the the application to access the data directly from the record. By eliminating these types of calls there would be fewer calls the application programmer would have to read and digest.

GetWRefCon Call # (not completed and in debate)

input: theWindow:LONG - pointer to window's record.
output: refCon:LONG - reserved LONG for application's use

See SetWRefCon.

WmgrResChg Call # (not completed)

input: newRes:WORD - 0 go to 16 color mode, 1 go to 4 color mode.
output: None.

Tell the Window Manager the application would like to change screen resolution. The Window Manager will make any interal adjustments needed and also adjust all window positions so their orgins keep the same spot on the screen.

ReFresh Call # (not completed)

input: None.
output: None.

Redraws the entire desktop and all the windows. Useful when the screen was clobbered or after changing screen resolutions.

Mac Functions in debate:

Here are some Mac functions that made or may not be needed in Cortland. Many of them were referred to by Inside Macintosh as "Normally you won't have to call this procedure" or "special circumstances". I could not think of any common uses for these calls and suggest they not be implemented unless someone comes up with a use. They all would require extra flags, code, programming time, and bugs. It may be that these calls will unintensionally appear from code in which case they can be included without further efford.

GetNewWindow - without resources this is meaningless.

SetWindowPic - extra little goody.
GetWindowPic - if **SetWindowPic** goes, so does this.

DragGrayRgn - let's see where this falls in, most likely it will be in, I just don't know what form.

Mac Functions waiting:

These functions are on hold and will be implemented when the Window Manager is further along.

InvalRect
InvalRgn
ValidRect
ValidRgn