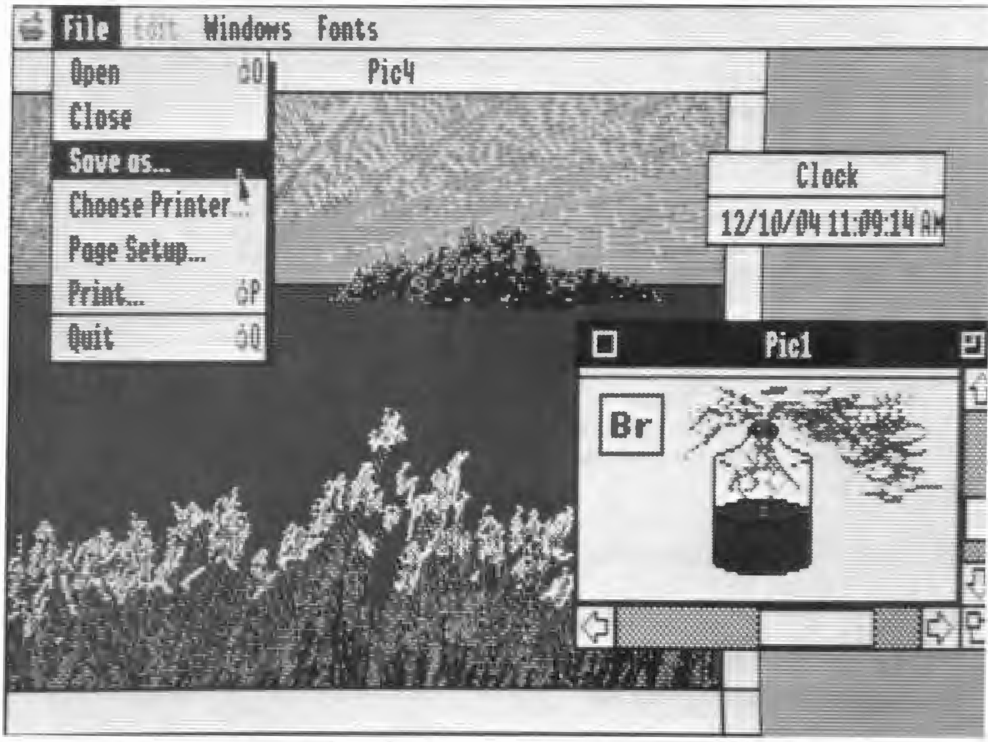
Apple® II

Apple IIGS® Toolbox Reference

Volume 2



Addison-Wesley Publishing Company, Inc.
Reading, Massachusetts Menlo Park, California New York Don Mills, Ontario Wokingham, England Amsterdam Bonn Sydney Singapore Tokyo Madrid San Juan

APPLE COMPUTER, INC.

Simultaneously published in the United States and Canada.

ISBN 0-201-17747-1
BCDEFGHIJ-DO-898
Second Printing, October 1988

Contents

Volume 1

Contents

Volume 2

Figures and tables

Chapter 21 Sound Tool Set 21-1

Chapter 22 Standard File Operations Tool Set 22-1

Chapter 23 Text Tool Set 23-1

Chapter 24 Tool Locator 24-1

Chapter 25 Window Manager 25-1

Chapter 16

QuickDraw II

Any time your desktop application needs to draw something, it uses **QuickDraw II** (and its extension, QuickDraw II Auxiliary). QuickDraw II is an adaptation and extension of the Macintosh toolbox component QuickDraw—it performs similar operations but has been enhanced to support Apple IIGS color.

QuickDraw II allows your application to

- Perform graphic operations easily and quickly
- Draw lines and shapes of various sizes and patterns
- Draw items in a variety of colors or gray scales
- Draw text in different fonts and with styling variations, such as italics and boldface

A preview of the QuickDraw II routines

To introduce you to the capabilities of QuickDraw II, all QuickDraw II routines are grouped by function and briefly described in Table 16-1. These routines are described in detail later in this chapter, where they are separated into housekeeping routines (discussed in routine number order) and the rest of the QuickDraw II routines (discussed in alphabetical order).

Table 16-1
QuickDraw II routines and their functions

Routine	Description
Housekeeping routines	
QDBootInit	Initializes QuickDraw II; called only by the Tool Locator—must not be called by an application
QDStartUp	Starts up QuickDraw II for use by an application
QDShutDown	Shuts down QuickDraw II when an application quits
QDVersion	Returns the version number of QuickDraw II
QDReset	Resets QuickDraw II; called only when the system is reset—must not be called by an application
QDStatus	Indicates whether QuickDraw II is active
Global environment routines	
GetStandardSCB	Returns a copy of the standard scan line control byte (SCB)
SetMasterSCB	Sets the master SCB to a specified value
GetMasterSCB	Returns a copy of the master SCB
InitColorTable	Returns a copy of the standard color table for the current mode
SetColorTable	Sets a specified color table to specified values
GetColorTable	Fills a specified color table with the contents of another color table
SetColorEntry	Sets the value of a color in a specified color table
GetColorEntry	Returns the value of a specified color in a specified color table
SetSCB	Sets the SCB to a specified value
GetSCB	Returns the value of a specified SCB
SetAllSCBs	Sets all SCBs to a specified value
SetSysFont	Sets a specified font as the system font
GetSysFont	Returns a handle to the current system font
ClearScreen	Sets the words in screen memory to a specified value
GrafOn	Turns on Super Hi-Res graphics mode
GrafOff	Turns off Super Hi-Res graphics mode
SetBufDims	Sets the size of the QuickDraw II clipping and text buffers, padding the values to permit large values of *chExtra* and *spExtra* and to allow for style modifications
ForceBufDims	Sets the size of the QuickDraw II clipping and text buffers, but doesn't pad the values in any way
SaveBufDims	Saves QuickDraw II's buffer-sizing information in an eight-byte record
RestoreBufDims	Restores QuickDraw II's internal buffers to the sizes described in the eight-byte record created by the SaveBufDims routine

Table 16-1 (continued)
QuickDraw II routines and their functions

Routine	Description
GrafPort routines	
OpenPort	Initializes specified memory locations as a standard GrafPort, allocates a new visible region and a new clipping region, and makes the GrafPort the current port
InitPort	Initializes specified memory locations as a standard port
ClosePort	Deallocates the clipping and visible regions in a port
SetPort	Makes a specified port the current GrafPort
GetPort	Returns a pointer to the current GrafPort
SetPortLoc	Sets the current port's *locInfo* record to specified location information
GetPortLoc	Gets the current port's *locInfo* record and puts it at the specified location
SetPortRect	Sets the current GrafPort's port rectangle to the specified rectangle
GetPortRect	Returns the current GrafPort's port rectangle
SetPortSize	Changes the size of the current GrafPort's port rectangle
MovePortTo	Changes the location of the current GrafPort's port rectangle
SetOrigin	Adjusts the contents of the port rectangle and the bounds rectangle so the upper left corner of the port rectangle is set to the specified point
SetClip	Copies a specified region into the clipping region
GetClip	Copies the clipping region to a specified region
ClipRect	Changes the clipping region of the current GrafPort to a rectangle that is equivalent to a specified rectangle
Pen and pattern routines	
HidePen	Decrements the pen level
ShowPen	Increments the pen level
GetPen	Returns the pen location
SetPenState	Sets the pen state in the GrafPort to specified values
GetPenState	Returns the pen state from the GrafPort to a specified location
SetPenSize	Sets the current pen size to a specified pen size
GetPenSize	Returns the current pen size to a specified location
SetPenMode	Sets the current pen mode to a specified pen mode
GetPenMode	Returns the pen mode from the current GrafPort
SetPenPat	Sets the current pen pattern to a specified pen pattern
GetPenPat	Copies the current pen pattern from the current GrafPort to a specified location
SetSolidPenPat	Sets the pen pattern to a solid pattern using the specified color
SetPenMask	Sets the pen mask to a specified mask
GetPenMask	Returns the pen mask to a specified location
SetBackPat	Sets the background pattern to a specified pattern
GetBackPat	Copies the current background pen pattern from the current GrafPort to a specified location
SetSolidBackPat	Sets the background pattern to a solid pattern using a specified color
SolidPattern	Sets a specified pattern to a solid pattern using a specified color
PenNormal	Sets the pen state to the standard state; pen location is not changed
MoveTo	Moves the current pen location to a specified point
Move	Moves the current pen location by specified horizontal and vertical displacements

(continued)

Table 16-1 (continued)
QuickDraw II routines and their functions

Routine	Description
Font routines	
SetFont	Sets the current font to a specified font
GetFont	Returns a handle to the current font
GetRomFont	Fills a specified buffer with information about the font in ROM
SetFontID	Sets the *fontID* field in the GrafPort
GetFontID	Returns the *fontID* field of the GrafPort
GetFontInfo	Returns information about the current font in a specified record
GetFGSize	Returns the size of the font globals record
GetFontGlobals	Returns information about the current font in a specified record
SetFontFlags	Sets the font flags word to a specified value
GetFontFlags	Returns the current font flags word
SetTextFace	Sets the text face to a specified value
GetTextFace	Returns the current text face
SetTextMode	Sets the text mode to a specified value
GetTextMode	Returns the current text mode
SetSpaceExtra	Sets the *spExtra* field in the GrafPort to a specified value
GetSpaceExtra	Returns the value of the *spExtra* field from the GrafPort
SetCharExtra	Sets the *chExtra* field in the GrafPort to a specified value
GetCharExtra	Returns the *chExtra* field from the GrafPort
SetForeColor	Sets the *fgColor* field (foreground color) in the GrafPort to a specified value
GetForeColor	Returns the value of the current *fgColor* field (foreground color) from the GrafPort
SetBackColor	Sets the *bgColor* field (background color) in the GrafPort to a specified value
GetBackColor	Returns the value of the current *bgColor* field (background color) from the GrafPort
Miscellaneous GrafPort routines	
SetClipHandle	Sets the *clipRgn* handle field in the GrafPort to a specified value
GetClipHandle	Returns a copy of the handle to the clipping region
SetVisRgn	Copies a specified region into the visible region (but does not change the *visRgn* field of the GrafPort)
GetVisRgn	Copies the contents of the visible region into a specified region
SetVisHandle	Sets the *visRgn* field in the GrafPort to a specified value
GetVisHandle	Returns a copy of the handle to the visible region
SetPicSave	Sets the *picSave* field in the GrafPort to a specified value
GetPicSave	Returns the value of the *picSave* field of the GrafPort
SetRgnSave	Sets the *rgnSave* field in the GrafPort to a specified value
GetRgnSave	Returns the value of the *rgnSave* field of the GrafPort
SetPolySave	Sets the *polySave* field in the GrafPort to a specified value
GetPolySave	Returns the value of the *polySave* field of the GrafPort
SetGrafProcs	Sets the *grafProcs* field of the current GrafPort to a specified value
GetGrafProcs	Returns the pointer to the *grafProcs* record associated with the GrafPort
SetUserField	Sets the *userField* field in the GrafPort to a specified value
GetUserField	Returns the value of the *userField* field of the GrafPort
SetSysField	Sets the *sysField* field in the GrafPort to a specified value—must not be called by an application
GetSysField	Returns the value of the *sysField* field of the GrafPort

Table 16-1 (continued)
QuickDraw II routines and their functions

Routine	Description
Line drawing routines	
LineTo	Draws a line from the current pen location to a specified point
Line	Draws a line from the current pen location to a new point specified by the horizontal and vertical displacements
Rectangle drawing routines	
FrameRect	Draws the frame of a specified rectangle using the current pen mode, pen pattern, and pen size
PaintRect	Paints the interior of a specified rectangle using the current pen mode and pen pattern
EraseRect	Erases the interior of a specified rectangle by filling it with the background pattern
InvertRect	Inverts the pixels in the interior of a specified rectangle
FillRect	Fills the interior of a specified rectangle with a specified pen pattern
Round rectangle drawing routines	
FrameRRect	Draws the frame of a specified round rectangle using the current pen mode, pen pattern, and pen size
PaintRRect	Paints the interior of a specified round rectangle using the current pen mode and pen pattern
EraseRRect	Erases the interior of a specified round rectangle by filling it with the background pattern
InvertRRect	Inverts the pixels in the interior of a specified round rectangle
FillRRect	Fills the interior of a specified round rectangle with a specified pen pattern
Region drawing routines	
FrameRgn	Draws the frame of a specified region using the current pen mode, pen pattern, and pen size
PaintRgn	Paints the interior of a specified region using the current pen mode and pen pattern
EraseRgn	Erases the interior of a specified region by filling it with the backgound pattern
InvertRgn	Inverts the pixels in the interior of a specified region
FillRgn	Fills the interior of a specified region with a specified pen pattern
Polygon drawing routines	
FramePoly	Draws the frame of a specified polygon using the current pen mode, pen pattern, and pen size
PaintPoly	Paints the interior of a specified polygon using the current pen mode and pen pattern
ErasePoly	Erases the interior of a specified polygon by filling it with the background pattern
InvertPoly	Inverts the pixels in the interior of a specified polygon
FillPoly	Fills the interior of a specified polygon with a specified pen pattern

(continued)

Table 16-1 (continued)
QuickDraw II routines and their functions

Routine	Description
Oval drawing routines	
FrameOval	Draws the frame of a specified oval using the current pen mode, pen pattern, and pen size
PaintOval	Paints the interior of a specified oval using the current pen mode and pen pattern
EraseOval	Erases the interior of a specified oval by filling it with the background pattern
InvertOval	Inverts the pixels in the interior of a specified oval
FillOval	Fills the interior of a specified oval with a specified pen pattern
Arc drawing routines	
FrameArc	Draws the frame of a specified arc using the current pen mode, pen pattern, and pen size
PaintArc	Paints the interior of a specified arc using the current pen mode and pen pattern
EraseArc	Erases the interior of a specified arc by filling it with the background pattern
InvertArc	Inverts the pixels in the interior of a specified arc
FillArc	Fills the interior of a specified arc with a specified pen pattern
Pixel transfer routines	
ScrollRect	Shifts the pixels inside the intersection of a specified rectangle, visible region, clipping region, port rectangle, and bounds rectangle
PaintPixels	Transfers a region of pixels
PPToPort	Transfers pixels from a source pixel map to the current port and clips the pixels to the current visible region and clipping region
Text drawing and measuring routines	
DrawChar	Draws a specified character at the current pen location and updates the pen location
DrawText	Draws specified text at the current pen location and updates the pen location
DrawString	Draws a specified Pascal-type string at the current pen location and updates the pen location
DrawCString	Draws a specified C string at the current pen location and updates the pen location
CharWidth	Returns the character width, in pixels (pen displacement), of a specified character
TextWidth	Returns the character width, in pixels (pen displacement), of specified text
StringWidth	Returns the sum of all character widths, in pixels (pen displacements), of a specified Pascal-type string
CStringWidth	Returns the sum of all the character widths, in pixels (pen displacements), of a specified C string
CharBounds	Puts the character bounds rectangle of a specified character into a specified buffer
TextBounds	Puts the character bounds rectangle of specified text into a specified buffer
StringBounds	Puts the character bounds rectangle of a specified Pascal-type string into a specified buffer
CStringBounds	Puts the character bounds rectangle of a specified C string into a specified buffer

Table 16-1 (continued)
QuickDraw II routines and their functions

Routine	Description
Calculations with rectangles	
SetRect	Sets a specified rectangle to specified values
OffsetRect	Offsets a specified rectangle by specified displacements
InsetRect	Insets a specified rectangle by specified displacements
SectRect	Calculates the intersection of two rectangles and places the intersection in a destination rectangle
UnionRect	Calculates the smallest rectangle that contains both source rectangles and places the result in a destination rectangle
PtInRect	Detects whether the pixel below and to the right of a specified point is in a specified rectangle
Pt2Rect	Copies a specified point to the upper left corner of a specified rectangle and another point to the lower right corner of the rectangle
EqualRect	Indicates whether two rectangles are equal
NotEmptyRect	Indicates whether a specified rectangle is not empty
Calculations with points	
AddPt	Adds two specified points together and leaves the result in the destination point
SubPt	Subtracts the source point from the destination point and leaves the result in the destination point
SetPt	Sets a point to specified horizontal and vertical values
EqualPt	Indicates whether two points are equal
LocalToGlobal	Converts a point from local coordinates to global coordinates
GlobalToLocal	Converts a point from global coordinates to local coordinates
Calculations with regions	
NewRgn	Allocates space for a new region and initializes it to an empty region—this is the only way to create a new region
DisposeRgn	Deallocates the memory for a specified region
CopyRgn	Copies the region definition from one region to another
SetEmptyRgn	Destroys previous region information by setting a specified region to an empty region
SetRectRgn	Destroys previous region information by setting a specified region to a specified rectangle
RectRgn	Destroys previous region information by setting a specified region to a specified rectangle
OpenRgn	Allocates temporary space and starts saving lines and framed shapes for later processing as a region definition
CloseRgn	Completes the region definition process started by an OpenRgn call
OffsetRgn	Moves a region on the coordinate plane a specified distance
InsetRgn	Shrinks or expands a specified region
SectRgn	Calculates the intersection of two regions and places the intersection in a destination region

(continued)

Table 16-1 (continued)
QuickDraw II routines and their functions

Routine	Description
Calculations with regions	
UnionRgn	Calculates the smallest region that contains every point that is in either source region and places the result in a destination region
DiffRgn	Calculates the difference of two regions and places the difference in a destination region
XorRgn	Calculates the difference between the union and the intersection of two regions and places the result in a destination region
PtInRgn	Checks to see whether the pixel below and to the right of a specified point is within a specified region
RectInRgn	Checks whether a specified rectangle intersects a specified region
EqualRgn	Indicates whether two regions are equal
EmptyRgn	Indicates whether a specified region is empty
Calculations with polygons	
OpenPoly	Returns a handle to a polygon data structure that will be updated by future LineTo calls
ClosePoly	Completes the polygon definition process started with an OpenPoly call
KillPoly	Disposes of a specified polygon
OffsetPoly	Offsets a specified polygon by specified horizontal and vertical displacements
Mapping and scaling utilities	
MapPt	Maps a specified point from a source rectangle to a destination rectangle
MapRect	Maps a specified rectangle from a source rectangle to a destination rectangle
MapRgn	Maps a specified region from a source rectangle to a destination rectangle
MapPoly	Maps a specified polygon from a source rectangle to a destination rectangle
ScalePt	Scales a specified point from a source rectangle to a destination rectangle
Cursor-handling routines	
SetCursor	Sets the cursor to an image passed in a specified cursor record
GetCursorAdr	Returns a pointer to the current cursor record
HideCursor	Hides the cursor by decrementing the cursor level
ShowCursor	Shows the cursor by incrementing the cursor level
ObscureCursor	Hides the cursor until the mouse moves
InitCursor	Reinitializes the cursor
Miscellaneous QuickDraw II utilities	
Random	Returns a pseudorandom number in the range –32768 to 32767
SetRandSeed	Sets the seed value for the random number generator
GetPixel	Returns the pixel below and to the right of a specified point
GetAddress	Returns a pointer to a specified table
SetIntUse	Indicates to the cursor drawing code whether the code should use scan line interrupts
SetStdProcs	Sets up a specified record of pointers for customizing QuickDraw II operations

Drawing to the screen and elsewhere

QuickDraw II can draw to the screen or to other parts of Apple IIGS memory. In fact, printing a document with the Print Manager involves using QuickDraw II to "draw" your document into a memory buffer used by the Print Manager (see Chapter 15, "Print Manager," in Volume 1).

To get our bearings, we'll first consider where QuickDraw II draws. Then we'll briefly discuss how it draws, and finally look at what it draws.

Where QuickDraw II draws

The question of *where* QuickDraw II draws involves consideration of Apple IIGS memory (including screen memory) as well as QuickDraw II's own internal representation of its drawing universe. These are the main concepts for you to remember:

- Drawings are stored in Apple IIGS memory as *pixel images,* ordered collections of bytes that represent rectangular arrays of pixels. Screen memory contains a special pixel image—its contents are displayed on the computer's monitor.
- QuickDraw II draws its text and graphic objects on an abstract two-dimensional mathematical surface called the *coordinate plane.* Points on a plane are much easier to visualize and manipulate than addresses in memory. Locations on the QuickDraw II coordinate plane are related to pixel-image locations by specific *location information* supplied to QuickDraw II.
- QuickDraw II draws most objects within the context of *graphic ports.* A port is a complete drawing environment and defines, among other things, a specific part of memory and a specific rectangular area on the coordinate plane where drawing can occur. There can be many ports open at a time—some for drawing to the screen, some for drawing to other parts of memory. Different ports' drawing spaces may be separate from each other, or they may overlap.
- QuickDraw II can be made to *clip,* or constrain its drawing, to within limits of arbitrary size, shape, and location.
- By manipulating two independent sets of coordinates (*global coordinates* and *local coordinates*), an application can easily control both what gets drawn inside a port's drawing space and where on the screen or other pixel image that drawing space appears.

Coordinate plane

QuickDraw II locates every action it takes in terms of coordinates on a two-dimensional grid, as shown in Figure 16-1. The grid is QuickDraw II's **coordinate plane;** coordinates on the plane are integers ranging from –16K to +16K in both the X and Y directions. The point (0,0), therefore, is in the middle of the grid. Note also that grid values increase to the right and *downward* on the plane; this is different from what you might be used to, but it is the direction and order in which video scan lines are drawn.

Units on the grid are in terms of **pixels.** Thus a 10 x 10 square on the coordinate plane is equivalent to a rectangle 10 pixels by 10 pixels on the display screen (this would not be a square, of course, because Apple IIGS pixels are not square). Only a very small portion of the coordinate plane can be displayed on the screen at any one time—the plane is 32,000 pixels on a side, and the screen can show a maximum of 640 pixels by 200 pixels at a time. The small black square near the center of the grid in Figure 16-1 shows the approximate size of the screen (in 320 mode) compared with the coordinate plane.

Warning

QuickDraw II must not be asked to draw outside the coordinate plane. Commands to draw outside this space will produce unpredictable results. They won't generate errors.

❖ *Macintosh programmers:* This conceptual drawing space is not the same as that for QuickDraw on the Macintosh. On the Macintosh, the drawing space is 64K by 64K pixels centered around 0,0, which makes the boundary coordinates –32K,–32K and 32K,32K.

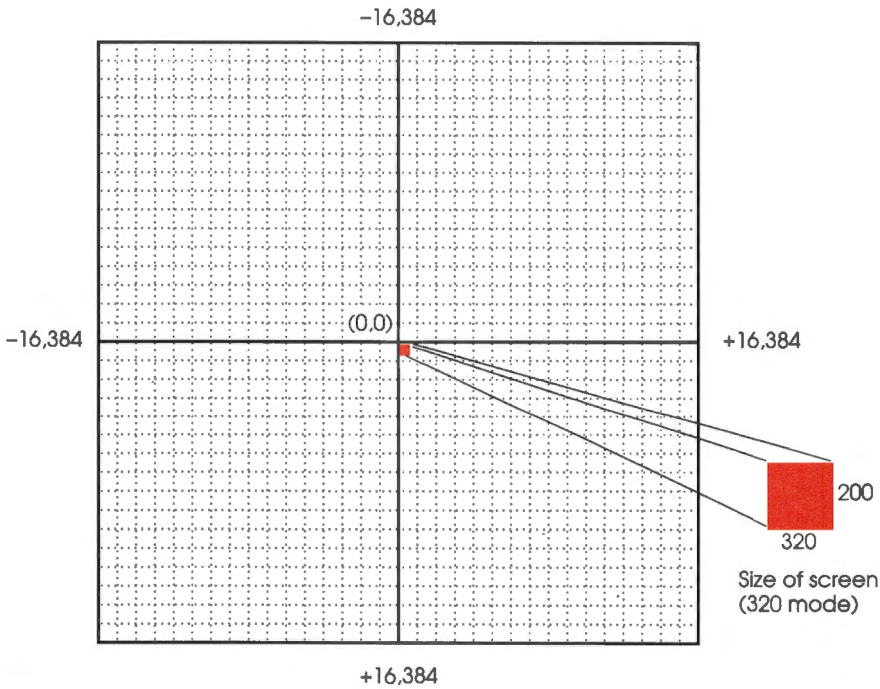


Figure 16-1
QuickDraw II coordinate plane

To understand how QuickDraw II does its drawing, we need to consider how it represents some basic graphic elements. On the coordinate plane, grid lines are considered to be infinitely thin. A point is defined as the intersection of two grid lines, so it also has no dimensions. Pixels, on the other hand, have a definite size; they are thought of as falling *between* the lines of the grid. The smallest element QuickDraw II can draw is a pixel, so if it were to draw a point at the location (3,3) on the coordinate plane, it would have to draw a single pixel. But which one? The point is between four pixels.

QuickDraw defines the pixel corresponding to each point on the plane as the pixel immediately *below and to the right* of the point, as shown in Figure 16-2.

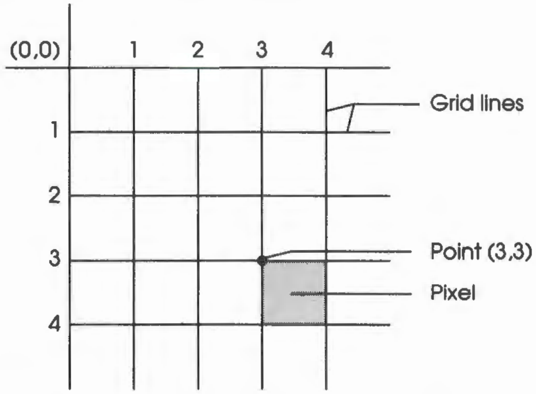


Figure 16-2
Grid lines, a point, and a pixel on the coordinate plane

Pixel images and the coordinate plane

A **pixel image** is an area of memory that contains a graphic image. The image is organized as a rectangular grid of pixels occupying contiguous memory locations. Each pixel has a value that determines what color in the graphic image is associated with that pixel.

❖ *Macintosh programmers:* QuickDraw II's pixel images are similar to Macintosh QuickDraw's bit images. The major difference is that a pixel is described by more than a single bit.

QuickDraw II draws to the coordinate plane. However, the coordinate plane is really just an abstract concept. Inside the Apple IIGS, drawing actually occurs by modifying pixel images—that is, by modifying the contents of certain memory locations. In particular, drawing something visible on the screen involves modifying the pixel image corresponding to screen memory.

The data structure that ties the coordinate plane to memory is the *locInfo* (for *location information*) record. The *locInfo* record tells QuickDraw II where in memory to draw, how the pixel image in that part of memory is arranged, and what that image's position on the coordinate plane is. The structure of the *locInfo* record definition is shown in Figure 16-3, and each field is described in more detail following the figure.

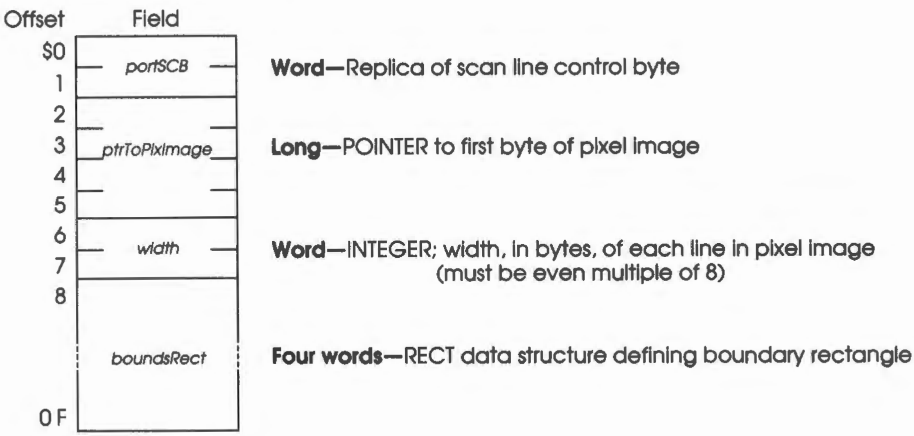


Figure 16-3
The *locInfo* record

The fields are as follows:

- The *portSCB* (a replica of the scan line control byte) tells QuickDraw II how many bits per pixel there are in this image—two for 640 mode, four for 320 mode. The scan line control byte and the differences between 640 mode and 320 mode are discussed further in the section "Drawing in Color" in this chapter.
- The *ptrToPixImage* field contains the **image pointer;** that is, the memory address of the image. It points to the first byte of the pixel image, which contains the first (upper leftmost) pixel.
- The *width* field specifies the **image width;** that is, the width (in bytes, not pixels) of each line in the pixel image. QuickDraw II needs to know this so it can tell where each new row in the image starts. (The image width must be an even multiple of eight bytes.)
- The *boundsRect* (for **boundary rectangle**) is a RECT data structure defining the rectangle that maps the pixel image onto the coordinate plane. The top left point in the rectangle corresponds to the first pixel in the image. The bottom right of the rectangle describes the extent of the pixel image (as far as QuickDraw II is concerned).

The image pointer, the image width, and the boundary rectangle defined by the *boundsRect* are shown in Figure 16-4.

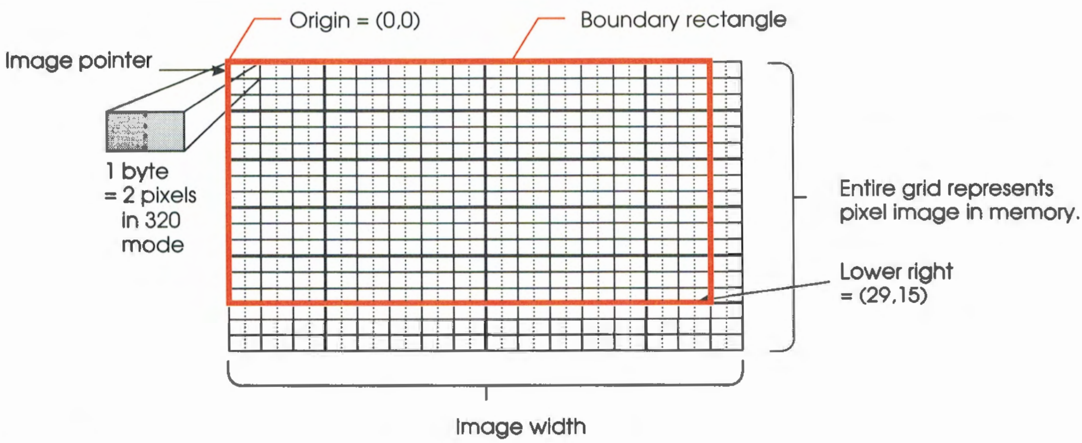


Figure 16-4
Pixel image and boundary rectangle

❖ *Note:* Remember that what separates one pixel image from another is where in memory it is stored, not where on the QuickDraw II coordinate plane its boundary rectangle happens to be. You can think of each pixel image as having its own private copy of the entire coordinate plane to play with, so even if two pixel images have overlapping coordinate plane locations, there won't be any conflict between them if they occupy completely different parts of computer memory.

GrafPort, port rectangle, and clipping

Most drawing takes place in conjunction with a data structure called a **GrafPort** (for **graphic port**). Each GrafPort contains a complete specification of a drawing environment, including the location information (*locInfo* record) described earlier. In addition to the location information, a GrafPort contains three other fields that restrict where drawing in a pixel image can take place: the port rectangle, clipping region, and visible region.

The **port rectangle** (as specified by the *portRect* field in the GrafPort) is a rectangle on the coordinate plane. Any drawing through a GrafPort occurs only inside its port rectangle. When you look at a window on the screen in a desktop application, its interior (everything but its frame) corresponds to a port rectangle. Windows are described further in Chapter 25, "Window Manager."

The port rectangle can coincide with the boundary rectangle, or it can be different. You can think of it as a movable opening that allows access to all or part of the pixel image. As Figure 16-5 shows, QuickDraw II can draw only where the boundary rectangle and port rectangle overlap.

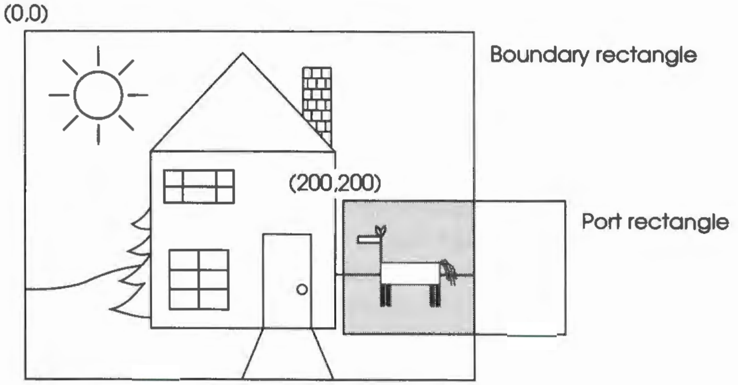


Figure 16-5
Boundary rectangle/port rectangle intersection

The **clipping region** (as specified by the *clipRgn* field in the GrafPort) is provided for an application to use. When a GrafPort is opened or initialized, the clipping region is set to the entire coordinate plane (effectively preventing any clipping from occuring). The program can use the clipping region any way it wants. Drawing to a pixel image through a GrafPort occurs only inside the clipping region.

The **visible region** (as specified by the *visRgn* field in the GrafPort) is normally maintained by the Window Manager. An application can have multiple windows on the screen, each one associated with a GrafPort. Windows can overlap, and each port's visible region represents the parts of the window that are visible.

❖ *Note:* When using GrafPorts that are not associated with windows, it is up to your application to maintain the visible region. When a GrafPort is opened or initialized, its *visRgn* field is set to be equal to the *portRect* field. It will stay that way until your program changes it.

In summary, drawing occurs in a pixel image only in the intersection of the boundary rectangle, port rectangle, clipping region, and visible region.

Global and local coordinate systems

In QuickDraw II's universe, everything is positioned in terms of coordinates on the plane. However, if you think of multiple open windows on the screen, you can see that there are at least two different ways in which you might want to locate objects:

- You may want to specify where windows appear on the screen (when they are moved, for example).
- You may want to specify where objects appear within windows (when scrolling, for example) independently of where on the screen the windows may be.

The toolbox needs **global coordinates** whenever more than one GrafPort share the same pixel image; the global coordinates tell QuickDraw II exactly where every port rectangle is compared with every other one. The global coordinate system for each GrafPort is that in which the boundary rectangle for its pixel image has its **origin** at (0,0) on the coordinate plane. In QuickDraw II, the origin of a rectangle is its upper left corner. For drawing to the screen, you can think of global coordinates as screen coordinates, where the top left corner of the screen is the point (0,0).

However, each port also has its own **local coordinate** system. For example, when drawing into a port, you can think in terms of distance from the port rectangle's origin rather than the boundary rectangle's origin. By defining the port rectangle as starting at (0,0), you can base all your drawing commands on distance in from the left edge and down from the top of the port rectangle.

That's convenient for drawing in a window, but local coordinates offer more convenience than that. They aren't constrained to a value of (0,0) for the port rectangle origin—you can set them to any coordinate-plane value. Why would you want to? Because of the way drawing commands work.

Suppose you are using a window to display portions of a document that is larger than the port rectangle in size—a fairly common occurrence. You are using drawing commands that draw the entire document, and you know that's no problem because the drawing will be automatically clipped to the port rectangle. But how do you control which part of the document shows through in your window? You do it by adjusting local coordinates.

For example, consider a document that has (0,0) as its origin. All QuickDraw II's drawing commands are based on the current port's local coordinate system. So if location (0,0) in your GrafPort's local coordinates correspond to the port rectangle's upper left corner, any time you draw your document into that port, its upper left corner will be displayed. If you define your local coordinates differently, different parts of your document will appear in the window. Thus, you can think of local coordinates as document coordinates—the upper left corner of the document that the port displays is the local coordinate origin—as shown in Figure 16-6.

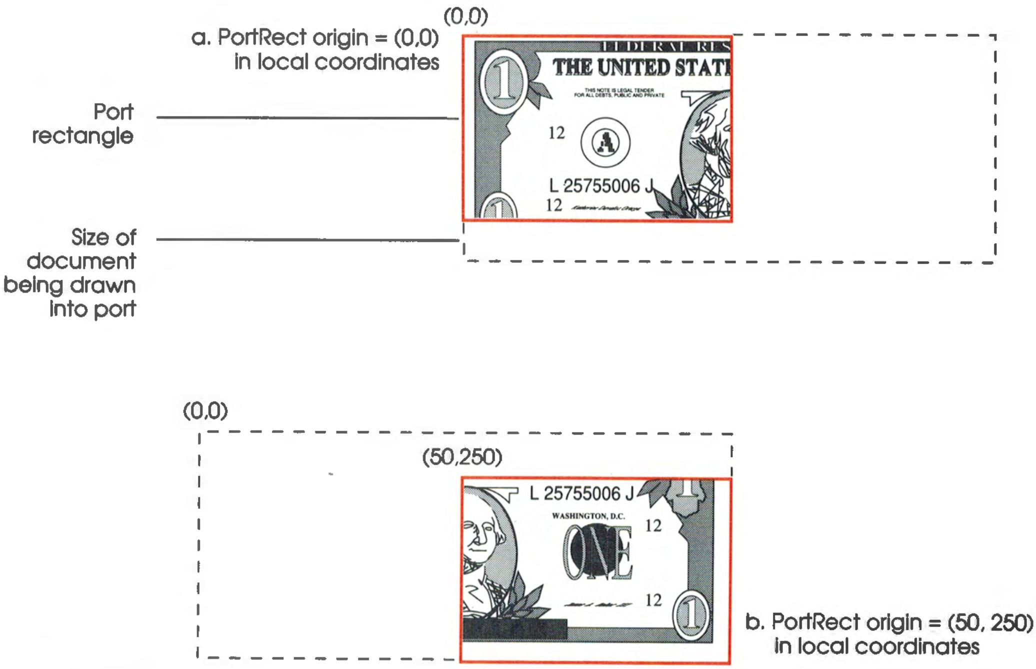


Figure 16-6
Drawing different parts of a document by changing local coordinates

❖ *Note:* When the local coordinates of a GrafPort are changed, the coordinates of the GrafPort's boundary rectangle and visible region are similarly recalculated, so the port will not change its relative position on the screen or its relation to other open ports on the screen.

However, when the local coordinates are changed, the GrafPort's clipping region and pen location are not changed—that is, they would appear to shift right along with the image that is being viewed in the port. It makes sense to have the pen (which is used to modify the image being viewed) and the clipping region (which is used to mask off parts of the image being viewed) stay with the image.

How QuickDraw II draws

The way QuickDraw II draws any of its objects depends on the drawing environment specified in the current GrafPort. Each GrafPort record includes location and clipping information (described earlier), information about the graphics pen (described next), information about any text that will be drawn (described in the section "Drawing Text" in this chapter), and other information, such as the patterns to draw with.

Drawing pen

Each open port has its own drawing **pen.** The pen controls where and how drawing (of both text and graphics) occurs. It has several characteristics (which can be set by the application) that control this.

Pen location: The pen has a coordinate-plane location (in local coordinates). The pen location is used only for drawing lines and text—other shapes are drawn independently of pen location.

Pen size: The pen is a rectangle that can have almost any width or height. Its default size is 1 x 1 (pixels). If either the width or the height is set to 0, the pen will not draw.

Pen pattern: The pen pattern is a repeating array (eight pixels by eight pixels) that is used like ink in the pen. Wherever the pen draws, the pen pattern is drawn in the image. The pattern is always aligned with the coordinate plane so that adjacent areas of the same pattern drawn at different times will blend in a continuous manner.

Background pattern: The background pattern is an array similar to the pen pattern. The process of **erasing** is that of drawing with the background pattern.

Drawing mask: The drawing mask is an eight-bit by eight-bit pattern that is used to mask, or screen off, parts of the pattern as it is drawn. Only those pixels in the pattern aligned with an *on* (1) bit in the mask are drawn. Figure 16-7 shows how a mask affects drawing with a pattern.

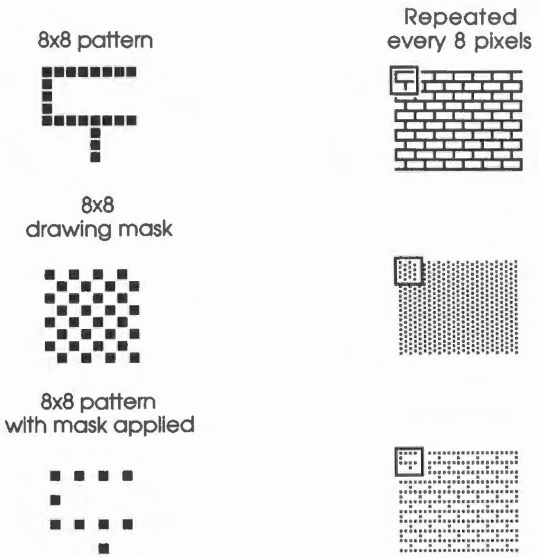


Figure 16-7
Drawing with pattern and mask

Note that drawing with a mask in which every bit has the value 1 is like drawing with no mask at all—all pen pixels are passed through to the image. Likewise, drawing with a mask that is all 0's is like not drawing at all—all pen pixels are blocked.

Pen mode: The pen mode specifies one of eight Boolean operations (`modeCopy`, `notCopy`, `modeOr`, `notOR`, `modeXOR`, `notXOR`, `modeBIC`, and `notBIC`) that determine how the pen pattern is to affect an existing image. When the pen draws, QuickDraw II compares pixels in the existing image with their corresponding pixels in the pattern and then uses the pen mode to determine the value of the resulting pixels. For example, with a pen mode of `modeCopy`, the existing pixels' values are ignored—a solid black line is black regardless of the image already on the plane. With a pen mode of `notXOR`, the bits in each pen pixel are inverted, then combined in an exclusive-OR operation with the bits in each corresponding existing pixel. Figure 16-8 shows a filled rectangle drawn over an existing circle, in both `modeCopy` and `notXOR` mode.

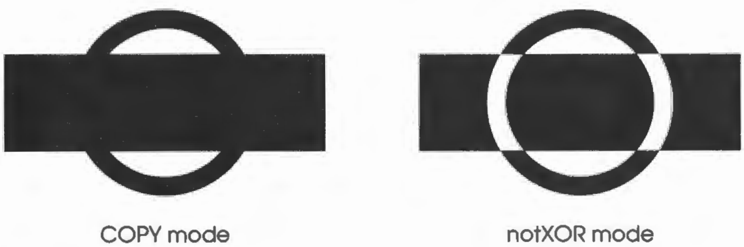


Figure 16-8
How pen mode affects drawing

The uses for the different pen modes are shown in Table 16-2. For more detail on how to use the modes, see the section "SetPenMode" in this chapter.

Table 16-2
Pen modes

Mode	Description
`modeCopy`, `notCopy`	Copy source (or NOT source) to destination. The `modeCopy` mode is the typical drawing mode. For text, the fully colored text pixels (both foreground and background) are copied into the destination.
`modeOR`, `notOR`	Overlay (OR) source (or NOT source) and destination. Use `modeOR` to nondestructively overlay new images on top of existing images and `notOR` to overlay inverted images. For text, the fully colored text pixels (both foreground and background) are ORed with the destination.
`modeXOR`, `notXOR`	Exclusive-or (XOR) pen with destination. Use these modes for cursor drawing and rubber-banding. If an image is drawn using `modeXOR`, the appearance of the destination at the image location can be restored merely by drawing the image again in `modeXOR`. For text, the fully colored text pixels (both foreground and background) are XORed with the destination.
`modeBIC`, `notBIC`	Bit Clear (BIC) pen with destination ((NOT pen) AND destination). Use this mode to explicitly erase (turn off) pixels, often prior to overlaying another image. You can use `notBIC` to display the intersection of two images. For text, the fully colored text pixels (both foreground and background) are BICed with the destination.

Basic drawing functions

QuickDraw II draws lines using the current pen size, pen pattern, drawing mask, and pen mode. It draws other shapes (rectangles, rounded-corner rectangles, ovals, arcs, polygons, and regions) in five different ways:

- **Framing** uses the current pen size, pen pattern, drawing mask and pen mode to draw an outline of the shape.
- **Painting** uses the current pen pattern, drawing mask, and pen mode to fill the interior of the shape.
- **Erasing** uses the current background pattern and drawing mask to fill the interior of the shape.
- **Inverting** uses the drawing mask to invert the pixels in the interior of the shape.
- **Filling** uses a specified pattern and the drawing mask to fill the interior of the shape.

QuickDraw II draws text as described in the section "Drawing Text" in this chapter.

What QuickDraw II draws

QuickDraw II can draw a number of graphic objects into a pixel image. It draws text characters in a variety of monospaced and proportional fonts, with styling variations that include italics, boldface, underlining, outlining, and shadowing. It draws straight lines of any length, width, and pattern. It draws hollow or pattern-filled rectangles, circles, and polygons. It draws elliptical arcs and filled wedges, irregular shapes, and collections of shapes. It also draws pictures—combinations of these simple shapes. Figure 16-9 summarizes QuickDraw II's graphic objects.

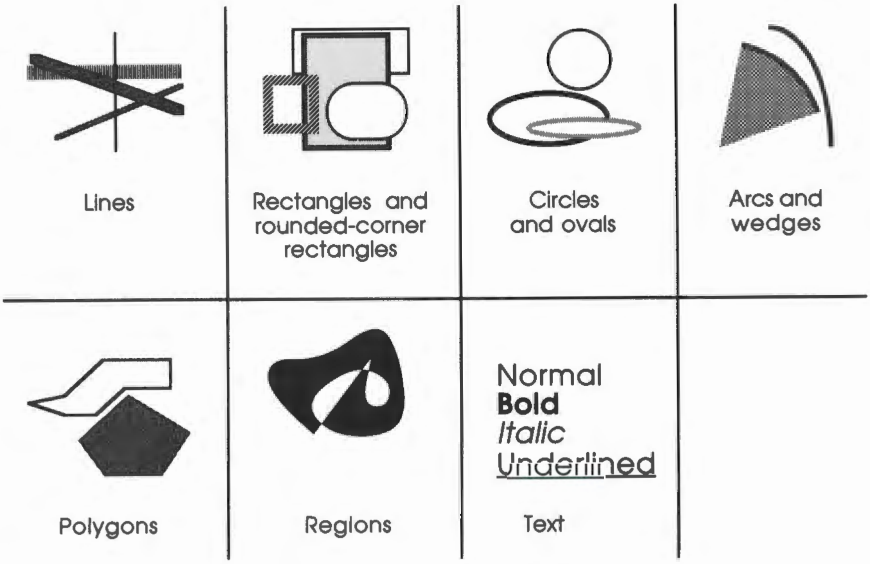


Figure 16-9
What QuickDraw II draws

Points and lines

A **point** is represented mathematically by its Y and X coordinates—these are two integers.

Important
QuickDraw II's data structure that defines a point has the vertical coordinate first: (y,x) rather than (x,y).

A **line** is represented by its ends—two points, or four integers. Like a point, a line is infinitely thin. When drawing a line, QuickDraw II moves the top left corner of the pen along the mathematical trajectory from the current pen location to the destination location. The pen hangs below and to the right of the trajectory, as illustrated in Figure 16-10.

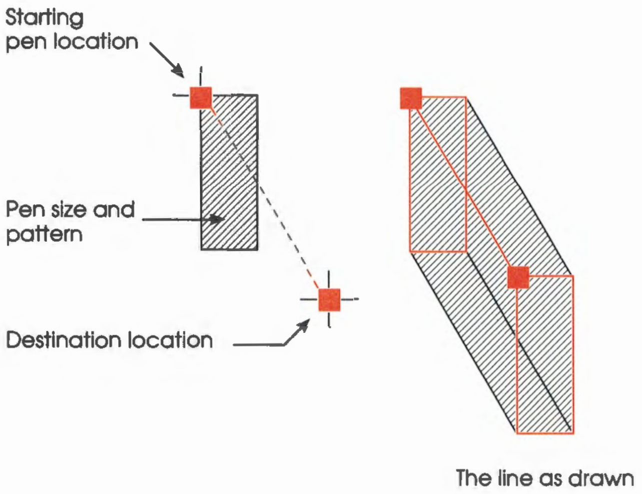


Figure 16-10
Drawing a line

Before drawing a line, you can use QuickDraw II calls to set the current pen location and other characteristics, such as pen size, mode, and pattern.

Rectangles

A **rectangle** is also represented by two points—its upper left and lower right corners. The borders of a rectangle are infinitely thin. Rectangles are fundamental to QuickDraw II; there are many functions for moving, sizing, and otherwise manipulating rectangles.

The pixels associated with a rectangle are only those within the rectangle's bounding lines. Thus, the pixels immediately below and to the right of the bottom and right-hand lines of the rectangle are not part of it.

Important
QuickDraw II's RECT data structure has coordinates in the following order: top, left, bottom, right. Thus, the defining coordinates for the rectangle in Figure 16-11 are (1,2,7,6). This order may be different from what you are used to, but it is consistent with the (y,x) ordering of points.

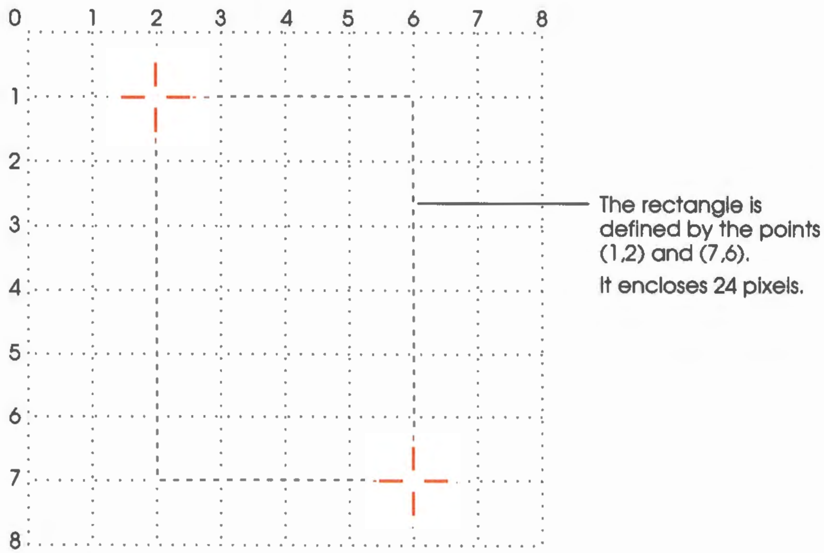


Figure 16-11
Rectangle

Rectangles may have square or rounded corners. The corners of **rounded-corner rectangles** (Figure 16-12) are sections of ovals (discussed next); they are specified by an oval height and an oval width.

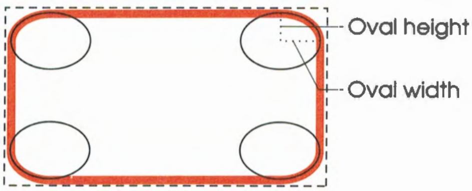


Figure 16-12
Rounded-corner rectangle

Circles, ovals, arcs, and wedges

Ellipses and portions of ellipses form another class of shapes drawn by QuickDraw II. An **oval** (Figure 16-13) is an ellipse; it is defined in the same way as a rectangle, with the exception that QuickDraw II is told to draw the ellipse enscribed within the rectangle rather than the rectangle itself. If the enclosing rectangle is a square, the resulting oval is a circle.

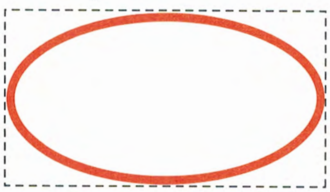


Figure 16-13
Oval

❖ *Pixel shape:* Remember, Apple IIGS pixels are not square. A perfect circle (or square) on the screen will have unequal horizontal and vertical dimensions in terms of pixels.

An **arc** (Figure 16-14) is a portion of an oval defined by the oval's enclosing rectangle and by two angles (the beginning and the end of the arc) and measured clockwise from vertical.

If an arc is painted, filled, inverted, or erased, it becomes a **wedge;** its fill pattern extends to the center of the enclosing rectangle, within the area defined by the beginning and ending angle lines.

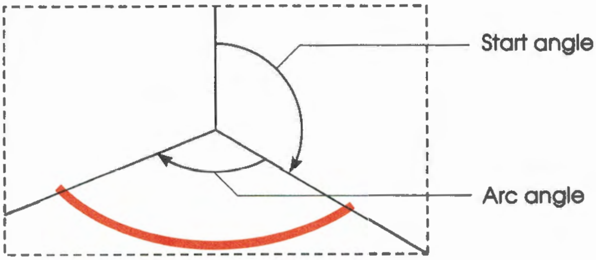


Figure 16-14
Arc

Polygons

A **polygon** (Figure 16-15) is any sequence of connected lines. You define a polygon by moving the pen to the starting point of the polygon and drawing lines from there to the next point, from that point to the next, and so on.

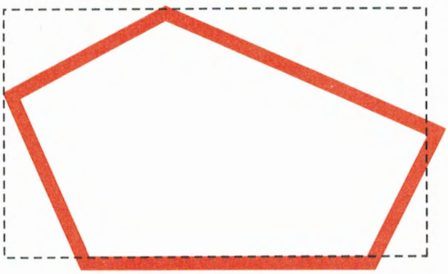


Figure 16-15
Polygon

Polygons are not treated exactly the same as other closed shapes, such as rectangles. For example, when QuickDraw II draws (*frames*) a polygon, it draws outside the actual boundary of the polygon because the line-drawing routines draw below and to the right of the pen locations. When it paints, fills, inverts, or erases a polygon, however, the fill pattern stays within the boundary of the polygon. If the polygon's ending point isn't the same as its starting point, QuickDraw II adds a line between them to complete the shape.

Regions

A region is another fundamental element of QuickDraw II, one that can be considerably more complex than a line or a rectangle. A **region** (Figure 16-16) is defined as a collection of shapes or lines (or other regions) whose outline is one or more closed loops. Your application can draw, erase, move, or manipulate regions the way it does any other QuickDraw II shapes.

You can define regions by drawing lines, framing shapes, manipulating existing regions, and equating regions to rectangles or other regions.

Regions are particularly important to the Window Manager, which must keep track of often irregularly shaped, noncontiguous portions of windows in order to know when to activate the windows or what parts of them to update.

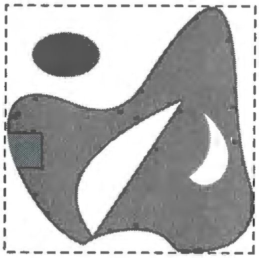


Figure 16-16
Region

Pictures

A **picture** is a collection of QuickDraw II drawing commands. Its data structure consists of little more than the stored commands. QuickDraw II plays the commands back when the picture is reconstructed with a DrawPicture call. A complex mechanical drawing produced from an Apple IIGS drafting program might be saved as a single QuickDraw II picture.

Pictures can be used to transfer data between applications via the Clipboard. See Chapter 20, "Scrap Manager."

Drawing text

QuickDraw II doesn't draw only graphic images—it also does all text drawing for desktop applications. As an application writer, you can easily control the placement, size, style, font, and color of display text with QuickDraw II calls.

Your program can provide QuickDraw II with text in the following formats:

- **Character:** A single ASCII character at a time
- **Pascal string:** A length byte followed by a sequence of ASCII characters
- **C string:** A sequence of ASCII characters terminated by a 0 byte ($00)
- **Text block:** A number of ASCII characters in a buffer, with the number specified separately

QuickDraw II draws the text in the same format in which it receives it. It draws each character at the current *pen location,* with the current *font,* using the current *text mode,* with the current character *style,* and using the current *foreground* and *background* colors. After text is drawn, the pen position is updated.

Simple text manipulation

This section introduces the text concepts you will need to know about for most applications. For most applications, you won't need to know anything more about fonts than is presented in this section. If you're writing an application that lets the user choose from a selection of fonts, or if you're developing an application that requires a specific font, you'll also need to know about the Font Manager. See Chapter 8, "Font Manager," in Volume 1.

A **font** is a collection of graphical and numerical information representing a set of characters. The graphical part of the font is called the **font strike** and consists of all the images of the characters, placed one after another. (The font strike in a IIGS font is stored in a one-bit-per-pixel format.) By convention, no blank space is left between the character images in the font strike; when text is drawn, both the space left between characters and the positioning of characters are determined by several tables of numerical information that are also part of the font. For the precise format of a IIGS font, see the section "Font Definition" in this chapter.

QuickDraw II always displays and measures text using the current font (whose handle is found in the *fontHandle* field of the current GrafPort).

A font has a **base line** (a horizontal line that runs throught the font strike), an **ascent** (the number of rows of pixels of the font above the base line), and a **descent** (the number of rows below). Each character in a font has a **character image** (the piece of the font strike that represents the character, using a bit set to 1 to represent the character's foreground pixels), a **character origin** (a point on the base line used to position the character with respect to the current pen position), and a **character width** (the number of pixels QuickDraw II will advance the pen position after it draws the character).

These concepts are illustrated in Figure 16-17.

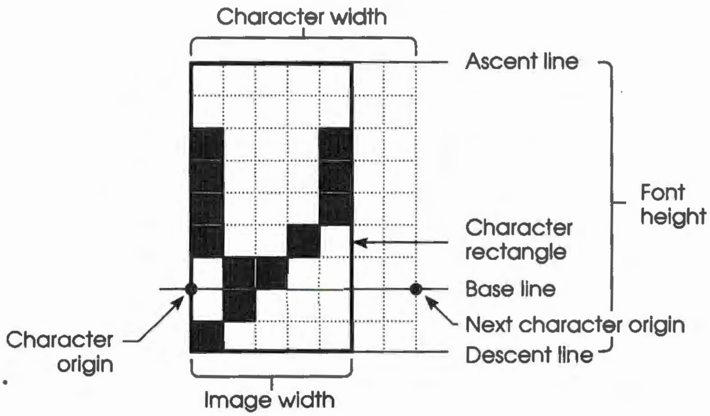


Figure 16-17
Character

When a character is drawn, it's placed so its character origin coincides with the current pen position. The character image's 1 bits, as mentioned, determine the **foreground pixels.** (The 0 bits of the character image are background pixels, but they are not the character's only background pixels. The precise definition of background pixels is provided later in this section.)

After the character is drawn, the pen is automatically advanced by the character width. The next character drawn will have its character origin at this new pen position (if the pen hasn't been moved first). Characters drawn one after the other are thus strung out horizontally in the expected manner.

For most characters in most fonts, the character image will lie between the old pen position and the new one. In fact, the new pen position will usually be several pixels to the right of the rightmost pixels of the character image; this supplies the small amount of blank space between characters. However, some characters in some fonts may have foreground pixels that lie to the left of the old pen position or to the right of the new pen position (or both). This is called **kerning.** When kerning occurs, the character images of adjacent characters (that is, characters drawn one after another) may possibly overlap.

The **character bounds rectangle** determines the extent of the background pixels of a character. The character bounds rectangle, relative to a current pen position and starting from the character origin, extends as follows:

- As far up as the font's ascent
- As far down as the font's descent
- As far left as the current (*old*) pen position, or as far as the character's leftmost foreground pixel, whichever is farther left (it is the leftmost foreground pixel if the character kerns to the left)
- As far right as the subsequent (*new*) pen position, or as far as the character's rightmost foreground pixel, whichever is farther right (it is the rightmost foreground pixel if the character kerns to the right)

Because pen positions are points, not pixels, the phrase *as far left as the current pen position* means that it includes the pixels immediately to the right of the current pen position. Similarly, *as far right as the subsequent pen position* means that it includes the pixels immediately to the left of the subsequent pen position.

This defines the character bounds rectangle, as shown in Figure 16-18.

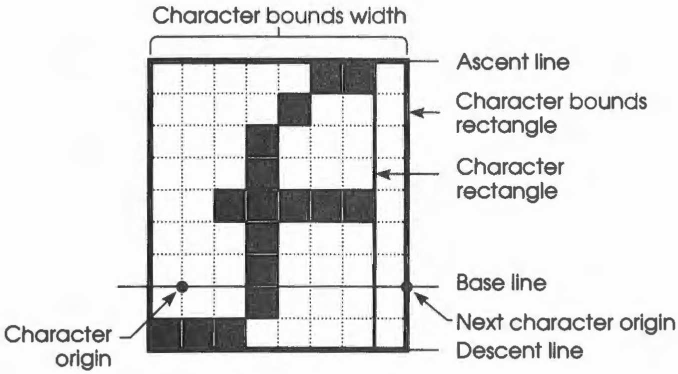


Figure 16-18
Character bounds rectangle

The character bounds rectangle contains all the foreground pixels of the character; that was the point of extending it as far as any kerning the character does in either direction. The **background pixels** of a character are defined to be all pixels in the character bounds rectangle that are not foreground pixels. Thus, the background pixels include all the pixels corresponding to 0 bits in the character image; in addition, they generally include those pixels extending from the old pen position to the new.

When QuickDraw II draws a character—say, using the pen mode modeCopy, with a foreground color of red and a background color of green—the foreground pixels are colored red and the background pixels are colored green. Our definition ensures that the background will at least go from pen position to pen position and that it will go far enough so no kerning foreground pixels will extend beyond the background.

The QuickDraw II routine CharBounds calculates the character bounds rectangle of a specified character in local coordinates based on the current pen position. Every pixel that could be affected by drawing the character is located inside the character bounds rectangle. This is different from the QuickDraw II call CharWidth, which simply returns the character width of a specified character—that is, the amount by which the pen position would be advanced if the character were drawn. The width of the character bounds rectangle is not the same thing as the character width; either one may be larger than the other, or one may even be 0 and the other nonzero. It is not necessary, however, that the widths be different, and for some characters, they may be the same.

❖ *Note:* Neither CharBounds nor CharWidth actually draws the character.

When QuickDraw II draws a string (whether a Pascal-type string, a C-type string, or a text block), it draws the individual characters of the string into an internal text buffer, advancing the position in the text buffer by the character width after each character and ORing the character images together whenever they overlap (as they may with kerning). Then the entire string is drawn into the destination pixel image, using the current text mode, foreground and background colors, and so on. The pen position is advanced by the sum of the character widths of all the characters in the string.

The QuickDraw II routines StringWidth, CStringWidth, and TextWidth return the amount the pen would be advanced if the specified string or text were to be drawn; that is, they return the sum of the character widths of all the characters in the string or text.

The QuickDraw II routines StringBounds, CStringBounds, and TextBounds return the smallest rectangle that would enclose all the foreground and background pixels of the string if it were drawn; in effect, they return the **string bounds rectangle.** This is the same as the UnionRect of all the individual character bounds rectangles (if the characters were drawn one after another).

Important

The rectangle is not necessarily the same as the rectangle you would get if you strung out the character bounds rectangles one after another, with the right edge of each touching the left edge of the next. Because of kerning, the character bounds rectangles of characters in a string may overlap.

❖ *Note:* Neither the bounds calls nor the width calls actually draw anything.

The bounds calls and width calls take into account any active style modifications, *chExtra* and *spExtra* values, and *fontFlags* settings that may affect either the area covered by foreground and background pixels or the amount the pen is advanced after text drawing. The QuickDraw II routine GetFontInfo takes into account the style modifications, but not the values of *chExtra*, *spExtra*, or *fontFlags*. The QuickDraw II routines GetFontLore and GetFontGlobals report on the current font as it exists in memory and do not take into account any of the other values mentioned.

In addition to the pen modes, which can be used for text, text can also be drawn in eight special text-only modes (four modes and their opposites). The uses for the different text modes are shown in Table 16-3. The opposite modes (`notForeCopy`, `notForeOR`, `notForeXOR`, and `notForeBic`) work the same way as the original modes, except that the foreground pixels are turned to background pixels and the background pixels are turned to foreground pixels before the operation is performed. For more detail on how to use the modes, see the section "SetTextMode" in this chapter.

Table 16-3
Text modes

Mode	Description
`modeForeCopy`, `notForeCOPY`	Copies only the foreground pixels into the destination—background pixels are not altered
`modeForeOR`, `notForeOR`	ORs only the foreground pixels into the destination—background pixels are not altered
`modeForeXOR`, `notForeXOR`	XORs only the foreground pixels into the destination—background pixels are not altered
`modeForeBIC`, `notForeBIC`	BICs only the foreground pixels into the destination; that is, inverts the source pixels and ANDs them with the destination—background pixels are not altered

If you need to know more about how fonts are drawn and constructed (if, for example, you want to write a font-editing application), see the section "Fonts and Text in QuickDraw II" in this chapter.

Drawing in color

The video display hardware of the Apple IIGS includes advanced color capabilities. Although tool calls make it unneccessary for you to manipulate the hardware directly, knowledge of a few background concepts will help you understand the way QuickDraw II manipulates the colors on the screen.

The Apple IIGS offers two Super-Hi-Res graphics modes. Both modes have 200 scan lines, but the scan lines differ in horizontal resolution—one mode has 320 pixels (the color of each specified by four bits), and the other has 640 pixels (the color of each specified by two bits). In changing from 320 mode to 640 mode, the horizontal resolution is doubled at the expense of dividing the color resolution by 4.

Both modes use a **chunky pixel** organization (in which the bits for a given pixel are contained in adjacent bits within one byte), as opposed to **bit planes** (in which adjacent bits in memory affect adjacent pixels on the screen). Therefore, the four bits of a pixel in 320 mode are in the same memory locations as the four bits of a pair of adjacent two-bit pixels in 640 mode.

Colors on the Apple IIGS are determined from **master color values,** which are mathematical combinations of the primary red, blue, and green hues available on a color monitor. A master color value is a two-byte number, formatted as shown in Figure 16-19.

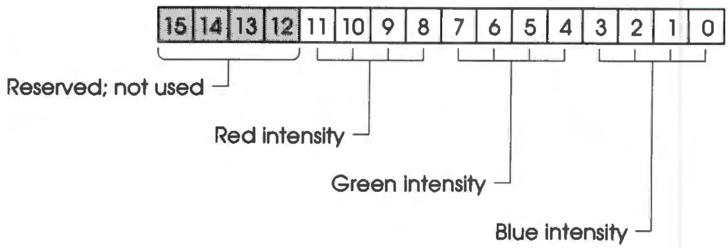


Figure 16-19
Master color value

A three-digit hexadecimal number can describe each master color, with one digit ($0–$F) for each primary color. Thus, a master color value of $000 denotes black, $FFF is white, $00F is the brightest possible blue, $080 is a medium-dark green, and so on. Because each primary color has 16 possible values, 4,096 colors are possible.

At any one time, the Apple IIGS uses only a small subset of all possible colors. An application does so by constructing one or more **color tables,** short lists of the available colors for any one pixel.

Color tables and palettes

Pixels contain only two or four bits, and it takes 12 bits to specify a master color value. Thus, applications cannot use master color values to directly specify pixel colors. Instead, the pixel value is a two- or four-bit offset into a color table.

A color table is a table of 16 two-byte entries. Each entry in the table is a master color value; any of the 4,096 possible color values may appear in any position in the color table. The colors available to the application, as specified in its color tables, constitute its **palette.**

Pixels in 320 mode are represented in memory by four-bit integers. For each pixel, that four-bit value is used as an a offset into a color table. With four bits, there are 16 possible pixel values, so the available colors for each pixel in 320 mode equals 16—the entire color table—as shown in Figure 16-20.

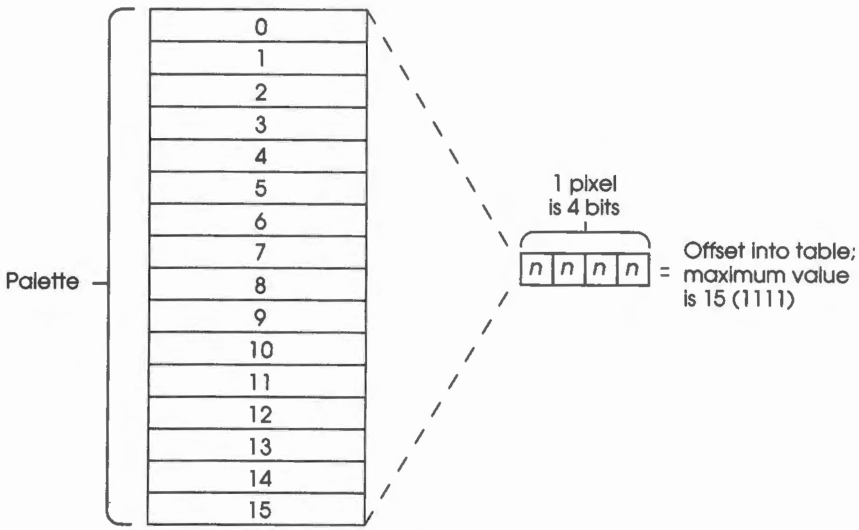


Figure 16-20
Accessing the color table in 320 mode

Pixels in 640 mode are represented in memory by two-bit integers. With two bits, there are only four possible pixel values to offset into the color table. To avoid limiting 640 mode to only four colors, however, each four adjacent pixels in 640 mode use four different parts of the same color table; a color table, then, consists of four **minipalettes,** which needn't have the same sets of master colors. Therefore, although each individual pixel in 640 mode can have one of only four colors, groups of four pixels can have a total of 16 colors from which to choose, as shown in Figure 16-21.

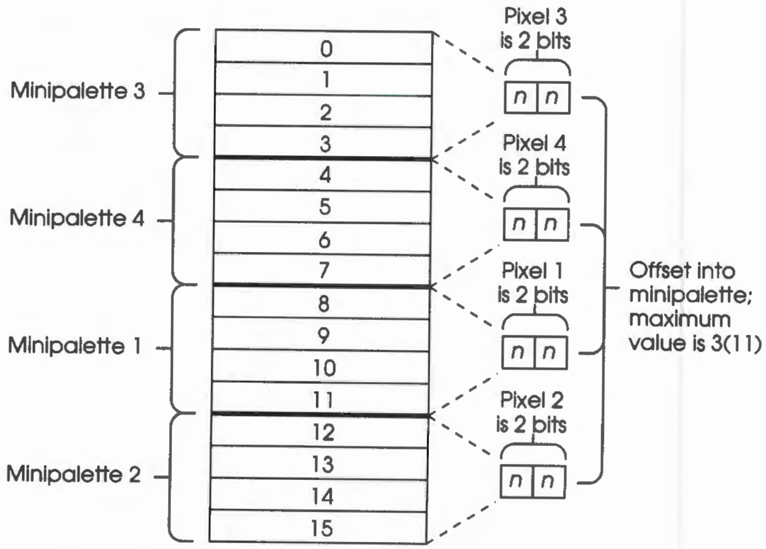


Figure 16-21
Accessing the color table in 640 mode

How to use this ability to create a large variety of colors is described in the section "Dithered Colors in 640 Mode" in this chapter.

Scan line control bytes

An application may construct as many as 16 different color tables to choose from. Each of the 200 scan lines in Super Hi-Res graphics can use any one of the 16 tables. For each scan line, a **scan line control byte (SCB)** decides which color table is active, as shown in Figure 16-22.

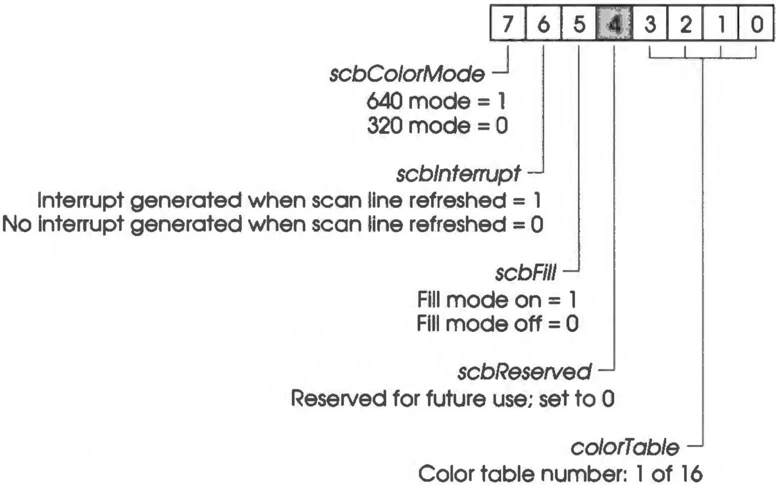


Figure 16-22
Scan line control byte

The SCB also controls screen display mode (320 or 640), interrupt mode, and fill mode.

Interrupt mode: Interrupts can be used to synchronize drawing with vertical blanking so pixels are not changed as they are being drawn (a pixel is drawn once every 1/60 of a second). Interrupts can also be used to change the color table before a screen is completely drawn. This allows a program to show more than 256 colors on the screen at once but costs the overhead of servicing the interrupt.

Fill mode: When fill mode is active, pixel values of 0 can be used to fill areas of color in 320 mode.

❖ *Note:* Fill mode works only in 320 mode.

A pixel with a numeric value of 0 serves as a placeholder indicating that the pixel should be displayed as the same color last displayed, as shown in Figure 16-23.

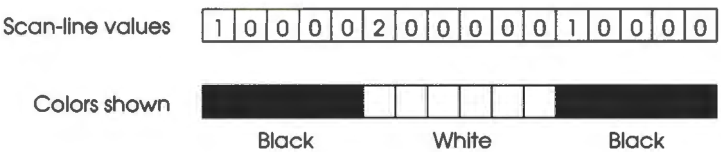


Figure 16-23
Fill mode example

Standard color palette in 320 mode

The standard palette (the default color table) for 320 mode is shown in Table 16-4.

Table 16-4
Standard palette in 320 mode

Offset	Color	Value	Offset	Color	Value
0	Black	000	8	Beige	FA9
1	Dark gray	777	9	Yellow	FF0
2	Brown	841	10	Green	0E0
3	Purple	72C	11	Light blue	4DF
4	Blue	00F	12	Lilac	DAF
5	Dark green	080	13	Periwinkle blue	78F
6	Orange	F70	14	Light gray	CCC
7	Red	D00	15	White	FFF

Note: "Offset" means position in the color table and "Value" means master color value, the hexadecimal value controlling the fundamental red-green-blue intensities.

Dithered colors in 640 mode

Only four colors are available for each pixel in 640 mode. But when small pixels of different colors are next to each other on the screen, their colors blend. For example, a black pixel next to a white pixel appears to the eye as a larger gray pixel. By cleverly choosing the entries in the color table, we can make more colors appear on the screen. This process is called **dithering.**

At the same time, to preserve the maximum resolution for displaying text, both black and white must be available for each pixel. This leaves only two remaining colors per pixel to choose from, which seems like a severe restriction. But with dithering, you can have 640-mode resolution for text and still display 16 or more colors if you are willing to resort to a few simple tricks.

Consider the following byte with four pixels in it:

Bit value	0 1	0 1	0 1	0 1
Pixel number	1	2	3	4

Each pixel has the value 1, which is an index into the second place in each of the color table's minipalettes (see Figure 16-21). So pixel 1's color is determined by entry 1 in minipalette 1, pixel 2's color is determined by entry 1 in minipalette 2, and so on.

If we use the standard 640-mode color table (shown in Table 16-5), pixels 1 and 3 will appear blue ($00F) and pixels 2 and 4 will appear red ($D00). The eye will average these colors and see violet.

Table 16-5
Standard palette in 640 mode

Offset	Color	Value	Minipalette offset	Offset	Color	Value	Minipalette offset
0	Black	000	0	8	Black	000	0
1	Blue	00F	1	9	Blue	00F	1
2	Yellow	FF0	2	10	Yellow	FF0	2
3	White	FFF	3	11	White	FFF	3
4	Black	000	0	12	Black	000	0
5	Red	D00	1	13	Red	D00	1
6	Green	0E0	2	14	Green	0E0	2
7	White	FFF	3	15	White	FFF	3

Note: The entries in the minipalettes for the standard 640-mode color table are set up so black and white appear in the same positions in each palette. This provides pure black and white at full 640 resolution, allowing crisper text display.

There are 16 different combinations of values a pair of pixels can assume in 640 mode, meaning that you can obtain 16 colors by dithering. To implement it, just make sure that the pattern you use for drawing or filling consists of a repeating array of four-bit (two-pixel) values.

Cursors

A **cursor** is a small image that appears on the screen and is controlled by a mouse. (The cursor appears only on the screen, never in an off-screen pixel image.) The **cursor record** contains the height and width of the cursor, the cursor image of the cursor, the mask controlling the appearance of the cursor, and a **hot spot** defining where the image of the cursor will be placed by the mouse, as shown in Figure 16-24.

Offset	Field	
$0	*cursorHeight*	**Word—**INTEGER; total number of horizontal slices in cursor
1		
2	*cursorWidth*	**Word—**INTEGER; number of words wide in single horizontal slice of cursor
3		
4	*cursorImage*	**x Bytes—**Array of words; cursor image (last word in each slice must be 0)
	cursorMask	**x Bytes—**Array of words; cursor mask (last word in each slice must be 0)
	hotSpotY	**Word—**INTEGER; Y coordinate of hot spot
	hotSpotX	**Word—**INTEGER; X coordinate of hot spot

Figure 16-24
Cursor record

❖ *Note:* Because of its variable size, the cursor record is not provided in the APW interface file.

The cursor appears on the screen as the size defined by the *cursorHeight* and *cursorWidth* fields in the cursor record. The appearance of each pixel is determined by the corresponding bits in the data mask and by the pixel under the cursor (that is, by the pixel already on the screen in the same position as this bit of the cursor). The image on the screen is obtained by ORing the mask with the destination and XORing that result with the cursor image.

The hot spot aligns a pixel in the cursor image with the mouse location. Thus, a hot spot of (0,0) is at the top left of the image, and a hot spot of (8,8) would be in the center of a cursor defined as 16 pixels wide and 16 pixels high.

The arrow cursor in 320 mode is defined as shown in the following assembly-language fragment:

```
dc i'11,4'                 ; Eleven slices by 4 words
dc h'0000000000000000'     ; Cursor image
dc h'0f00000000000000'
dc h'0FF0000000000000'
dc h'0FFF000000000000'
dc h'0FFFF00000000000'
dc h'0FFFFF0000000000'
dc h'0FFFFFF000000000'
dc h'0FFFFFFF00000000'
dc h'0FF0FF0000000000'
dc h'00000FF000000000'
dc h'0000000000000000'

dc h'FF00000000000000'     ; Mask image
dc h'FfF0000000000000'
dc h'FFFF000000000000'
dc h'FFFFF00000000000'
dc h'FFFFFF0000000000'
dc h'FFFFFFF000000000'
dc h'FFFFFFFF00000000'
dc h'FFFFFFFFF0000000'
dc h'FFFFFFFF00000000'
dc h'FFF0FFFF00000000'
dc h'00000FFF00000000'

dc i'1,1'                  ; Hot spot
```

Using QuickDraw II

This section discusses how the QuickDraw II routines fit into the general flow of an application and gives you an idea of which routines you'll need to use under normal circumstances. Each routine is described in detail later in this chapter.

QuickDraw II depends on the presence of the tool sets shown in Table 16-6 and requires that at least the indicated version of the tool set be present.

Table 16-6
QuickDraw II—other tool sets required

Tool set number		Tool set name	Minimum version needed
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0
$03	#03	Miscellaneous Tool Set	1.0

The first QuickDraw II call your application must make is QDStartUp. Conversely, when you quit your application, you must make the QDShutDown call.

All graphic operations are performed in GrafPorts. Before a GrafPort can be used, it must be opened with the OpenPort routine. Normally, you don't call OpenPort yourself—in most cases, your application will draw into a window you've created with Window Manager routines, and these routines call OpenPort to create the window's GrafPort. Similarly, a GrafPort's regions can be disposed of by the ClosePort routine. When you call the Window Manager to close or dispose of a window, it calls these routines for you.

In an application that uses multiple windows, each is a separate GrafPort. If your application draws into more than one GrafPort, you can call SetPort to set the GrafPort you want to draw in. At times, you may need to preserve the current GrafPort; you can do this by calling GetPort to save the current port, SetPort to set the port you want to draw in, and then SetPort again when you need to restore the previous port.

Some toolbox routines return or expect points that are expressed in a common, global coordinate system; others use local coordinates. For example, when the Event Manager reports an event, it gives the mouse location in global coordinates, but your application may need to know where the mouse location is in the window's local coordinates. The GlobalToLocal routine lets you convert global coordinates to local coordinates, and the LocalToGlobal routine lets you do the reverse.

The SetOrigin routine will adjust a GrafPort's local coordinate system. If your application performs scrolling, you'll use ScrollRect to shift the pixels of the image and then use SetOrigin to readjust the coordinate system after the shift.

You can redefine a GrafPort's clipping region with the SetClip or ClipRect routine. Just as GetPort and SetPort preserve the current GrafPort, GetClip and SetClip are useful for saving the GrafPort's clipping region while you temporarily perform other clipping functions. This is useful, for example, when you want to reset the clipping region to redraw the newly displayed portion of a document that's been scrolled.

The LineTo routine draws a line from the current pen location to a given point, and the Line routine draws a line as an offset from the current position. You can set the pen location with the MoveTo or Move routine; you can set other pen characteristics with SetPenSize, SetPenMode, and SetPenPat.

In addition to drawing text and lines, you can use QuickDraw II to draw a variety of shapes. Most of them are defined by a rectangle that encloses the shape. The following require you to call a series of routines to define them:

- To define a region, call the NewRgn routine to allocate space for it, then call OpenRgn, and then specify the outline of the region by calling routines that draw lines and shapes. End the region definition by calling CloseRgn. When you're completely done with the region, call DisposeRgn to release the memory the region occupies.
- To define a polygon, call the OpenPoly routine and then form the polygon by calling procedures that draw lines. Call ClosePoly when you're finished defining the polygon; call KillPoly when you're completely done with it.

You can perform the following graphic operations on rectangles, rounded-corner rectangles, ovals, arcs and wedges, regions, and polygons: framing, painting, erasing, inverting, and filling. These operations are described in the section "Basic Drawing Functions" in this chapter.

You'll use points, rectangles, and regions not only when drawing with QuickDraw II but also when using other parts of the toolbox. At times, you may find it useful to perform calculations on these entities. You can, for example, add and subtract points and perform a number of calculations on rectangles and regions, such as offsetting them, rescaling them, calculating their union or intersection, and so on.

❖ *Note:* When performing a calculation on entities in different GrafPorts, you need to adjust to a common coordinate system first by calling LocalToGlobal to convert to global coordinates.

Warning

QuickDraw II doesn't forgive certain kinds of programming errors. Results will be unpredictable if you make any QuickDraw II calls before initializing QuickDraw II, pass any bad handles to QuickDraw II, or make any QuickDraw calls with a bad port. Even an error code probably won't be returned if one of these kinds of errors is made, and your program may never get control again after one of these errors. Your application may not fail immediately; it may work for a while and then later fail for no apparent reason. You must be certain these types of errors can't occur in your program.

Fonts and text in QuickDraw II

This section contains a detailed description of the handling of fonts and text in QuickDraw II, including the definition of an Apple IIGS font.

❖ *Note:* Most application writers, most of the time, will not need any more information than is included in the previous sections about text handling in QuickDraw II. But if you are designing a font, writing a font editor, or using unusual fonts or an usually large variety of fonts, you'll need the information presented in this section.

❖ *Macintosh programmers:* The treatment of text drawing and text measurement on the Apple IIGS is similar to their treatment on the Macintosh. The IIGS font definition is similar to that of the Macintosh, and a simple conversion algorithm allows the IIGS to use any font developed for the Macintosh. Most Macintosh QuickDraw text calls are duplicated precisely in QuickDraw II. Any differences are due to one or more of the following:

- Some information has been added to the beginning of the font definition.
- Because Macintosh-like resources do not exist, the IIGS Font Manager performs differently from the Macintosh Font Manager.
- Some bounding box calls (TextBounds and its siblings) missing from Macintosh QuickDraw have been added.
- Some calls—DrawCString, CStringWidth, and the like—have been added to handle the C string data type (a sequence of characters terminated by a 0 byte).
- The Font Manager is not closely integrated with QuickDraw II. (The interaction between QuickDraw II and the Font Manager is quite different).
- QuickDraw II does not scale text. However, the Font Manager can scale fonts as required.

Font definition

An Apple IIGS font consists of a variable-length header, followed by a Macintosh font record (this embedded Macintosh font is referred to as the MF part of the IIGS font). The header is of variable length to allow extra information to be added in the future.

The MF part of a IIGS font is exactly like a Macintosh font, except for one thing—the high-order and low-order bytes of integers. The Macintosh's 68000 microprocessor stores integers with the high-order byte first (that is, high-order byte at lower memory location); the IIGS's 65816 microprocessor stores them with low-order byte first. In converting a Macintosh font to a IIGS font, the high-order and low-order bytes of each integer are swapped. This does not apply to the font strike (called *bitImage* in the font definition), which can be used as is.

Apple IIGS font definition

The IIGS font record is shown in Figure 16-25.

Offset	Field	
$0–1	*offseToMF*	**Word**—INTEGER; offset in words to Macintosh font part
2–3	*family*	**Word**—INTEGER; font family number
4–5	*style*	**Word**—INTEGER; style font was designed with
6–7	*size*	**Word**—INTEGER; point size
8–9	*version*	**Word**—INTEGER; version number of font definition
0A–0B	*fbrExtent*	**Word**—INTEGER; font bounds rectangle extent
0C		**x Bytes**—Additional fields, if any
	fontType	**Word**—INTEGER; font type—ignored on Apple IIgs
	firstChar	**Word**—INTEGER; ASCII code of first defined character
	lastChar	**Word**—INTEGER; ASCII code of last defined character
	widMax	**Word**—INTEGER; maximum character width
	kernMax	**Word**—INTEGER; maximum leftward kern
	nDescent	**Word**—INTEGER; negative of descent
	fRectWidth	**Word**—INTEGER; width of font rectangle
	fRectHeight	**Word**—INTEGER; height of font rectangle (font height)
	owTLoc	**Word**—INTEGER; offset in words to offset/width table
	ascent	**Word**—INTEGER; font ascent
	descent	**Word**—INTEGER; font descent
	leading	**Word**—INTEGER; leading
	rowWords	**Word**—INTEGER; width of font strike in words
	bitImage	**x Bytes**—Array (1...rowWords, 1...fRectHeight) of word; font strike
	locTable	**x Bytes**—Array (firstChar...lastChar + 2) of INTEGER; location table
	owTable	**x Bytes**—Array (firstChar...lastChar + 2) of INTEGER; offset/width table

Figure 16-25
Font definition

Apple IIGS font header fields

Some information about the font is contained in the header. Because fonts designed at a later time may include additional information that could be utilized by later versions of QuickDraw II, the header is of variable length. For upward and downward compatibility of QuickDraw II and IIGS fonts, the following two fields are particularly useful:

offseToMF: Offset, in words, from this field to the Macintosh font (MF) part included in the IIGS font (specifically, to the *fontType* field). The header is therefore 2 x *offseToMF* bytes long. In the version of QuickDraw II current at the time of this manual's publication, *offseToMF* = 6; thus, the header is 12 bytes. Future fonts may have longer headers that contain font information that can be utilized by future versions of QuickDraw II. To ensure that these improved fonts can be used by older versions of QuickDraw II, the *offseToMF* field provides a reliable jump over this extra font information to the start of the Macintosh part of the font. An older QuickDraw II will not be able to make use of new header fields added since it was implemented, but at least it will be able to find the information it can use.

version: Version number of the font definition under which the font was designed. By checking this field, later versions of QuickDraw II can avoid trying to find and use information not included in an older font. (Presumably a newer QuickDraw II, alerted by the version number to the lack of such information, would use some default or calculated values.) The font definition described in this manual is 1.1 ($0101).

Examples of extra information that may be included in later fonts and used by later versions of QuickDraw II include thickness of underline, slope of italicized letters, smearing extent of boldface, and the like (on the Macintosh, these are determined by the Font Manager).

The other header fields are

family: Integer identifying the font, regardless of size or style. This can be thought of as corresponding to the font's name—Courier or Geneva, for example.

style: Style in which the font was designed. For example, application writers and graphic designers may design italic or bold fonts for reasons of aesthetics or time performance. When QuickDraw II is asked to apply a certain style when drawing a character or string, it first checks this field. If the field indicates that the requested style is already part of the font, the drawing call will not apply the styling algorithm. This would, for example, prevent a preitalicized font from being reitalicized.

size: Relative measure of the size of fonts. The measure is analogous to the Macintosh point size; however, the actual font size is different from the true point size in a typographic sense.

fbrExtent: Maximum horizontal distance, in pixels, from the character origin to the far edge of any foreground or background pixel of any character in the font. (See the section "Font Bounds Rectangle" in this chapter for a more precise definition.)

Macintosh font part of an Apple IIGS font

A Macintosh font, or in this case the Macintosh font part of a IIGS font, consists of four sections:

1. A fixed-length record containing general information, such as font height and maximum character width.
2. The font strike (named *bitImage* in the font record definition), which is a pixel image containing the image of every character defined in the font, strung one after another. The pixel image is in a one-bit-per-pixel form. Its width, measured in words, is given by the *rowWords* field of the font record; its height, measured in pixels, is given by *fRectHeight*.
3. The location table (*locTable*), an array of integers that indicates for each defined character where its image in the font strike begins.
4. The offset/width table (*owTable*), an array of integers. For each character, the low-order byte of its entry in the offset/width table (the *character offset*) indicates how the character image to be drawn should be positioned with respect to the current pen location; the high-order byte indicates how far the pen should be advanced after the character is drawn (*character width*).

   This table is also used to identify characters not defined in a font. An *owTable* value of –1 ($FFFF) marks a missing character, which must be handled specially by the text calls.

A detailed description of the meanings and uses of these various fields and arrays is given in the sections that follow.

Characters

A character image is a rectangular array of bits, representing pixels. The on, or 1, bits are called the character foreground pixels.

The number of columns in a character image is called the **character image width,** or just the **image width.** Note that a character can have an image width of 0. For example, the space has a 0 image width; its character image consists of no pixels at all.

The **character rectangle** is a rectangle that encloses the character image. Its width is the image width of the character, which may vary from character to character in a font; its height is the character height, which is the same for all characters in a font.

Each character has a number associated with it. This number, called the **character width** and found in the offset/width table, is the number of pixels the pen position is to be advanced after the character is drawn. This is different from the image width, and the distinction between the two is important. For example, the space character has 0 image width but does have some positive character width, which determines the size of the space. Some characters have a nonzero image width but a 0 character width; an example of this is an umlaut, which is meant to be typed over a vowel. The umlaut is drawn first, and then the vowel is drawn with the same pen location. Characters with 0 character width are called **dead characters.**

Also associated with every character in a font are its base line and its character origin. The **base line** is a horizontal line that separates the image into two sets of rows, one set above and one below. (Remember that in QuickDraw II, as in QuickDraw, horizontal and vertical lines fall between pixels rather than running through them.) The position of the base line depends on the font's **ascent** and **descent** fields; it is chosen so there are *ascent* rows above it and *descent* rows below. The base line will be in the same horizontal position for every character in the font. Any foreground pixels of a character image that lie below the base line are collectively called the character's **descender.** Most characters don't have a descender, but in an average font, characters like *q* and *y* do.

The **ascent line** is the horizontal line just above the top row of a character; the **descent line** is the line just below the bottom row. These will be the same for every character in the font.

The **character origin** of a character is a point on the base line used to position the character for drawing. This point may be between pixels of the character image, to the right of them, or to the left. (Here, note that points lie between pixels, not on them.) Its location relative to the character image can be calculated by the character offset in the offset/width table, as detailed in later sections of this chapter. When the character is drawn, it is placed in the destination pixel image so that its character origin coincides with the current pen location.

For many letters, the character origin is located on the left edge of the character image so that, when the character is drawn, its leftmost foreground pixels fall just to the right of the pen. Sometimes the character origin is between pixels of the character image (or, rarely, entirely to the right of the image). When such a character is drawn, some of its pixels will fall to the left of the pen position. This is called kerning to the left. In such a case, the distance, in pixels, from the character origin to the left edge of the character is called the character's **leftward kern.**

When character-image pixels fall to the right of the new pen position after the character is drawn, the character is said to kern to the right. The *kernMax* field in a font is concerned only with kerning to the left. Kerning in either direction can cause letters to overlap each other. See Figures 16-26 and 16-27.

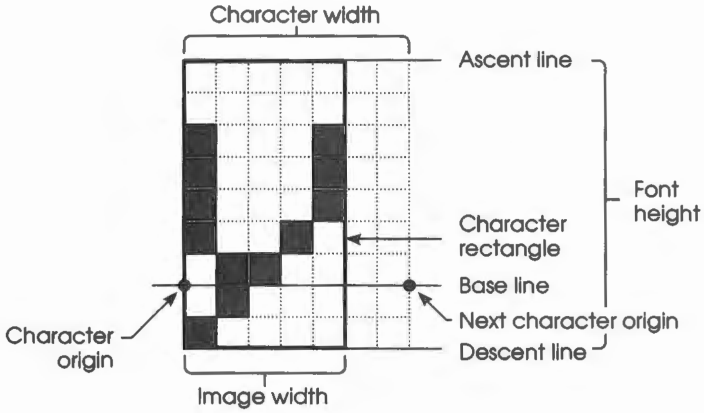


Figure 16-26
Character with no kerning

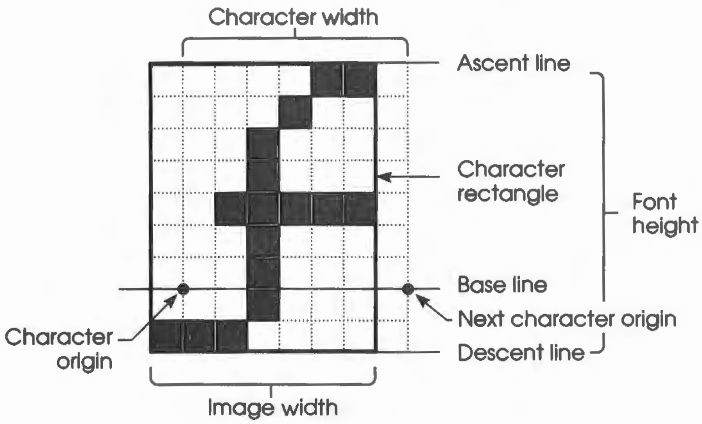


Figure 16-27
Character kerning left

Fonts

Font rectangle

Imagine all the defined characters of a font drawn so their character origins coincide. The result would be a black mess of foreground pixels. The smallest rectangle completely enclosing this mess is called the **font rectangle** (see Figure 16–28).

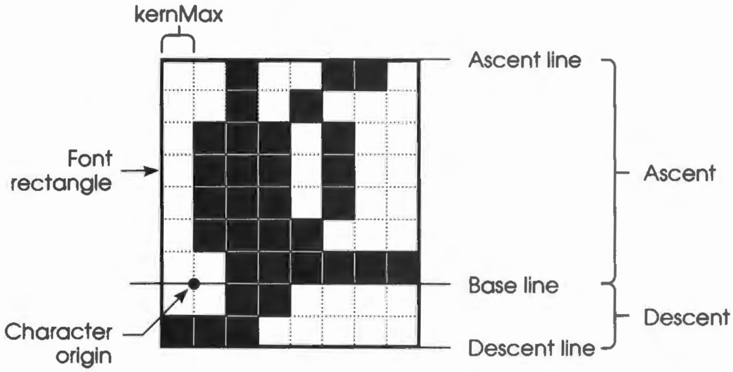


Figure 16-28
Font rectangle (simulated)

The fields of the font record that measure aspects of the font rectangle are described in the following list:

kernMax: Distance, in pixels, from the font rectangle's common character origin to the left edge of the font rectangle. If any character in the font actually kerns to the left, *kernMax* is represented as a negative number. If the character origin lies on the left edge of the font rectangle, *kernMax* is 0.

Most fonts fall into these two categories. However, in some fonts, the left edge of the font rectangle is one or more pixels to the right of the character origin. In such a case, *kernMax* is assigned a positive value, even though this bends the terminology a bit; for example, people do not usually say of a character that leaves two columns of blank pixels between the pen position and its image that it kerns to the left 2 pixels, or –2 pixels, or anything at all.

fRectWidth: Width, in pixels, of the font rectangle. Note that this may be more than the maximum character image width because the font rectangle's left and right extremes may come from different characters.

fRectHeight: Height, in pixels, of the font rectangle.

ascent: Number of pixel rows above the common base line in the font rectangle.

descent: Number of pixel rows below the base line in the font rectangle. Note that *fRectHeight* = *ascent* + *descent*.

nDescent: Negative of descent.

❖ *Note:* For typical fonts—those in which the font rectangle at least touches its character origin—*ascent* and *descent* will be non-negative, and *kernMax* and *nDescent* will be nonpositive. However, fonts can be designed without these restrictions.

fontType: QuickDraw II ignores this field.

firstChar: ASCII code of the first defined character in the font.

lastChar: ASCII code of the last defined character of the font.

widMax: Maximum character width (pen displacement) of any character in the font, measured in pixels.

owTLoc: Offset, in words, from this field to the font offset/width table (*owTable*). By adding 2 x *owTLoc* to the memory address of this field, you get a pointer to the *owTable*. There is no corresponding field for the location table in the font record; to get a pointer to the *locTable*, you must subtract 2 x (*lastChar* – *firstChar* + 3) from the *owTable* pointer.

leading: Recommended number of blank pixel rows between the descent row of one line of text and the ascent row of the next. Applications may use this or not, as they please.

rowWords: Width of the font strike, in words. This is discussed further in the next section, "Font Strike."

Font strike

The **font strike** (called *bitImage* in the font definition) is a one-bit-per-pixel pixel image consisting of the character images of every defined character in the font, placed sequentially in order of increasing ASCII code from *firstChar* to *lastChar* +1. The character images in the font strike abut each other; no blank columns are left between them; see Figure 16-29.

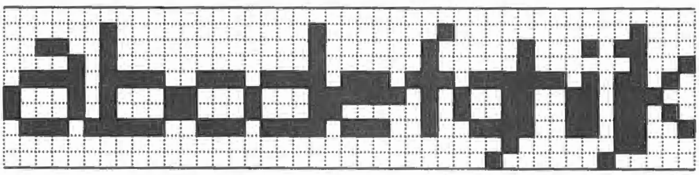


Figure 16-29
Part of a font strike

Because all the characters of a font have the same height, the font strike is just one long pixel image with no jumps or undefined stretches and with a height of *fRectHeight*. The strike is padded on the right, if necessary, with enough extra pixels on each row to make the row width a multiple of 16—that is, to make each row an integral number of words. This width, measured in words, is found in the *rowWords* field of the font record.

Defined versus undefined characters

Not every possible ASCII code must have a character image in the font strike. The font may leave some characters undefined; these are called **missing characters.** Every character with a code less than *firstChar* or greater than *lastChar* + 1 is undefined. There may be other undefined characters as well. The offset/width table (*owTable*) has an entry for every code from *firstChar* to *lastChar* + 2, inclusive. If a character's entry in the offset/width table is –1 ($FFFF), the character is undefined or missing.

Character code *lastChar* + 1 is a special case. Immediately following *lastChar* in the font strike is a character (known as the **missing symbol**) that is to be used in place of any missing character. This character must be present in the font strike. It has entries in the *locTable* and the *owTable*, and its entry in the *owTable* must not be –1. For all purposes, the missing symbol is a defined character with ASCII code *lastChar* + 1. In many fonts, the missing symbol is a hollow rectangle; in the current system font, it is a white-on-black question mark.

Whenever the QuickDraw II text-handling routines encounter a missing character—less than *firstChar*, greater than *lastChar* + 1, or having an *owTable* entry of –1—the routines immediately substitute the missing symbol for the character, using the missing symbol's character image, *locTable* entry, and *owTable* entry wherever needed.

Location table

The **location table** (*locTable*) is an array of integers with an entry for each character code from *firstChar* to *lastChar* + 2. It is used to find character images in the font strike. For each defined character, the *locTable* entry gives the distance, in pixels, from the beginning of the font strike to the beginning of the character's image in the font strike ("beginning," here, means left edge). This indicates where the character image starts. To see where it ends, take the next *locTable* entry (the beginning of the next character image) and subtract 1. Because the character images abut each other, this will give you the precise limits of the character image. The image width of a defined character with code C is *locTable*[C + 1] – *locTable*[C]. This may be 0.

For this scheme to work, two conditions must hold:

1. The *locTable* entry for an undefined character must be the same as the entry for the next defined character. This prevents undefined characters, which have no image in the strike, from interfering with the hunt for images of defined characters.

   Note that there always will be a next defined character because the missing symbol, which serves as a defined character, is tacked on at the end of the strike.

2. To get the character image for the missing symbol, there has to be an entry in the *locTable* following the missing symbol entry. For this reason *locTable*[*lastChar* + 2] is included and is set equal to the length of the font strike in pixels, ignoring the padding that is added to the font strike to align it to word boundaries.

Offset/width table

The offset/width table (*owTable*) is an array of integers with an entry for each character code from *firstChar* to *lastChar* + 2. If a character's entry is –1, the character is undefined (missing). Otherwise the entry's low-order and high-order bytes are the character width and character offset, respectively. Both are interpreted as numbers in the range 0–254 (255 is ruled out to avoid the case where both bytes are 255, giving an entry of –1, which would mark a missing character).

The character offset is used to calculate the position of character origin relative to the image in the following way: The offset is added to the font's *kernMax*. The result is the (horizontal) distance, in pixels, from the character origin to the left edge of the image. If the result is negative, then the origin is to the right of the image's left edge (the character kerns leftward). If the result is positive, the origin is to the left of the image's left edge. (A result of 0 means that the character origin sits on the left edge of the image). Because we already know that the character origin must lie on the base line (whose position is determined from ascent and descent), this precisely locates the origin.

If you draw the font rectangle and look at a particular character's character rectangle within it, the character offset is seen to be the offset, in pixels, between the left edge of the font rectangle and the left edge of the character rectangle. See Figure 16-30.

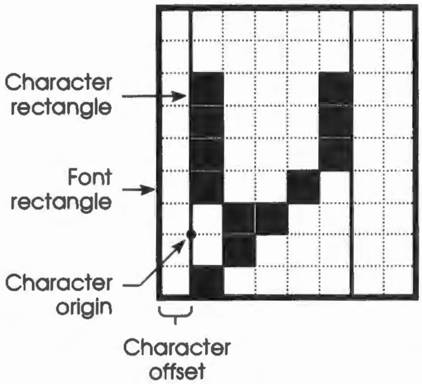


Figure 16-30
Character rectangle in font rectangle

The low-order byte of the *owTable* entry gives the character width, which is the distance, in pixels, the pen should be advanced to the right after the character is drawn. In applications, this distance can be affected by a number of calls, particularly SetCharExtra and SetSpaceExtra. There is, however, a general rule in QuickDraw II: any character whose character width (taken unmodified from the *owTable*) is 0 will not have that width changed by *chExtra, spExtra,* style modifications, nonproportionality, or any other effect. We assume that characters are given 0 width only for some very good reason.

Warning

Any modification, or any combination of modifications, that results in a character width of less than 0 or greater than 255 will wreak havoc with the drawing routines and is not allowed. This includes *chExtra, spExtra,* and style modifications, among others. QuickDraw II does not check for this condition. That is up to you.

The *lastChar* + 2 entry of the *owTable* is set to –1.

Character backgrounds

A character's foreground consists of all the *on* pixels (1 bits) in its image. The off pixels (0 bits) are part of the background. In QuickDraw II, the background is extended on the left to include any pixels that are to the left of the image's left edge but to the right of the character origin (and between the ascent and descent lines). On the right side, the background is extended to include any pixels (between ascent and descent) that are to the right of the image's right edge but to the left of the character origin of the next character (that is, to the left of the new pen position). Any new pixels added in this way are also considered background pixels. In other words, the foreground of a character consists of all 1 bits in its character image. The background consists of all 0 bits in the image plus all nonforeground pixels that are to the right of the character origin, to the left of the subsequent character origin (character origin + character width), above the descent line, and below the ascent line. In some cases, no extending is needed. If the character kerns to the left, no left extension is necessary; if it kerns to the right, no right extension is needed.

This is a very natural definition of background. If you're going to draw a green character that doesn't stretch entirely from the old pen position to the new and you have a red background, the red background will usually extend a little to the left and/or right of the character's image. This is what people generally want for a background. But in addition to this, when characters do kern, the background extends as far left or right as the kerning, so the kerned part of the character doesn't jut out past the character's background.

This brings us to the definition of the **character bounds rectangle** (Figure 16-31). It is the smallest rectangle enclosing all the foreground and background pixels of a character. It may be somewhat larger than the character rectangle, which encloses the image, because the bounds rectangle takes into account the character width (pen positions) as well as the image width. The width of a character's bounds rectangle is called the **character bounds width.** QuickDraw II includes calls for measuring character bounds rectangles and corresponding routines for strings, C strings, and text.

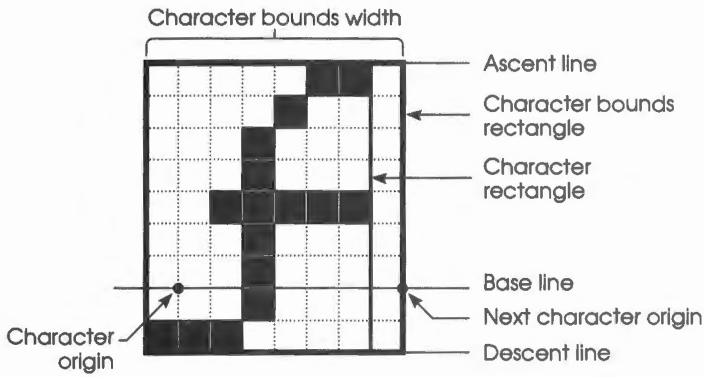


Figure 16-31
Character bounds rectangle

Font bounds rectangle

To get some new, useful measures for the width of a font, we define the font bounds rectangle. Imagine that, for all of a font's characters, the characters' bounds rectangles were drawn so all the character origins coincided. The resulting rectangle (more precisely, the rectangle that is the union of all these rectangles) is called the **font bounds rectangle.** This rectangle, illustrated in Figure 16-32, includes all pixels, foreground and background, of every character in the font. (Consequently, it may be bigger than the font rectangle, which is only guaranteed to include all the foreground pixels.)

We define *fbrWidth* to be the width of the font bounds rectangle, *fbrRightExtent* to be the distance from the common character origin to the right edge of the font bounds rectangle, and *fbrLeftExtent* to be the distance from the origin to the left edge (all distances measured in pixels and as positive numbers). Finally, we define *fbrExtent* to be the maximum of *fbrLeftExtent* and *fbrRightExtent*.

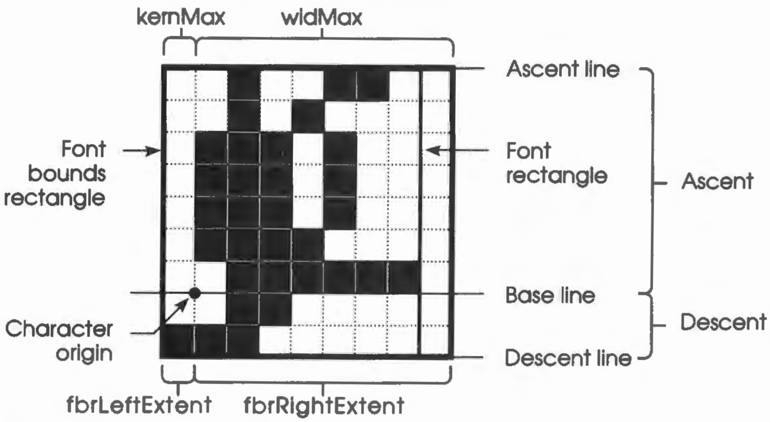


Figure 16-32
Font bounds rectangle

The *fbrExtent* value is the farthest possible horizontal distance from the pen location to the far edge of any pixel that can be altered by drawing any character in the font. In many ways, this is a more precise measure of the width of a font than *widMax* or *fRectWidth*.

It would seem from Figure 16-32 that *fbrLeftExtent* and *kernMax* are the same, or rather that *fbrLeftExtent* = –*kernMax*. This is true if, and only if, *kernMax* is 0 or negative; if *kernMax* is positive (that is, if every character image starts at least 1 pixel to the right of the character origin) then *fbrLeftExtent* is 0. This makes *fbrLeftExtent* easy to calculate. It would also seem as if *fbrRightExtent* is the same as *widMax*, but if any character kerns to the right beyond the reach of *widMax*, *fbrRightExtent* will be bigger than *widMax*.

❖ *Macintosh programmers:* The *fbrExtent* field is needed for safe handling of the text buffer. It is not included in the Macintosh font definition. If you are converting a Macintosh font to the IIGS, *fbrExtent* can be calculated by using the CharBounds call on character codes 0–255 and doing some simple arithmetic (the CharBounds call itself doesn't need a valid value for *fbrExtent*, so it can be called for the calculation). This has to be done only once for each font.

Drawing and the text buffer

Whenever a character or string is to be drawn, it is first drawn into the **text buffer,** a one-bit-per-pixel pixel image reserved for the private use of the QuickDraw II text-drawing calls. For strings, only those characters that have a chance of making it into the destination pixel image are actually drawn; the others, both to the left and to the right, contribute only to the cumulative pen displacement. Thus, there is no reason for an application to try to clip characters out of long strings unless it has a very fast way of doing so.

The text buffer is empty at the beginning of each drawing call. Successive characters of a string are drawn into it, with an internal text buffer pen incremented by the character width each time. Regardless of the ultimate text mode (*txMode* in the GrafPort), characters are drawn into the text buffer in OR mode. Thus, characters that kern into each other do not interfere destructively. (For this reason, with certain text modes, such as `modeForeXor`, the results you get if you put up a string one character at a time with DrawChar can be different from those you get if you put up the whole string with DrawString. In the case of DrawChar, overlapping characters may cancel out some pixels.)

Once the character or string is safely in the text buffer, any requested style modifications (underlining, bolding, and the like) are applied to it. Then the text buffer is transferred to the destination. Individual bits are replaced with two or four bits, depending on the chunkiness of the destination; the bit patterns used are the Grafport's *fgColor* for the 1 (foreground) bits and *bgColor* for the 0 (background) bits. During the transfer, the text image is clipped to the current clipping region. The surviving pixels are combined with the destination's pixels, according to whatever text mode is in use. If and when the result makes it to the screen, the bit patterns will be translated into colored pixels according to the current color map(s).

Controlling text display

Various QuickDraw II calls affect text display. Generally, the calls set some field of the current GrafPort that is used in the text-drawing process. Matching the calls that set the fields are corresponding calls that return the value of the fields.

SetForeColor, GetForeColor, SetBackColor, GetBackColor, SetTextMode, and GetTextMode deal with the GrafPort fields *fgColor, bgColor,* and *txMode,* whose effects were described earlier. The other display control calls deal with character spacing, style modifications, boldfacing, underlining, and font flag options, as described in the following sections.

Character spacing calls

SetCharExtra, GetCharExtra, SetSpaceExtra, GetSpaceExtra: These calls set and/or get the *chExtra* and *spExtra* fields in the GrafPort; those fields can alter the character widths (pen displacement) when characters are drawn. Each is a fixed number with a one-word integer part and a one-word fractional part.

The *chExtra* field was included because some fonts that look fine in 320 mode appear too closely spaced in 640 mode. Putting an extra pixel between letters seems to help in these cases.

The *chExtra* value is added to the character width of each character, except a dead character, as it is drawn. The *chExtra* value is not added to dead characters, which have a character width of 0.

Adding *chExtra* to a character width will give us character origin positions that have a fractional part. During any text-drawing call, QuickDraw II keeps track of this fractional part and, when drawing a character, rounds its character-origin position to the nearest integer (1/2—that is, $8000—is rounded up). The fractional part is not remembered after the call has been completed.

Commonly used to help in justifying text, *spExtra* works the same way as *chExtra* except that it is only applied to the space character. Note that here "space character" means ASCII $20 and nothing else. In particular, the nonbreaking space included in many fonts is unaffected by the *spExtra* field. The *spExtra* value is cumulative with *chExtra*.

These values are set by the QuickDraw II routines SetCharExtra and SetSpaceExtra (and can be fetched by GetCharExtra and GetSpaceExtra). In theory, the application can set *chExtra* and *spExtra* to any fixed value, even a negative one. However, any values that cause a character to have a character width of less than 0 or greater than 255 pixels will cause no end of trouble.

Style modification calls

SetTextFace, GetTextFace: These calls set and get the *txFace* field of the GrafPort, which determines the style to be applied to the text. At the time of publication, the following bits were defined:

Bit 0	Bold
Bit 1	Italic—available only if QuickDraw II Auxiliary is loaded and started up
Bit 2	Underline
Bit 3	Outline—available only if QuickDraw II Auxiliary is loaded and started up
Bit 4	Shadow—available only if QuickDraw II Auxiliary is loaded and started up

These styles may affect the character width (pen displacement), image width, ascent, and descent associated with a character or string. For example, boldfacing spaces characters farther apart and makes the characters thicker. These changes are reflected in the results returned by the width routines (CharWidth, StringWidth, and so on), the bounds routines (CharBounds, StringBounds, and so on), and the GetFontInfo routine.

Font flags option calls

SetFontFlags, GetFontFlags: The *fontFlags* field of the GrafPort is set by SetFontFlags and fetched by GetFontFlags. At the time of publication, only the last three bits (bits 2–0) of this word are defined; bits 15–3 are reserved and should be set to 0.

Bit 0: If bit 0 is set, the font is used as a nonproportional font; every character, except characters with character width 0, is given the same character width, namely *widMax* (the maximum character width field from the font definition).

When nonproportionality is in effect, *chExtra, spExtra,* style modifications, and so forth, are applied to the character width after it has been set to *widMax*.

Bit 1: As of Version 2.0, if bit 1 is set and bit 0 is not, every character in the font is given the same character width, just as occurs with the nonproportionality setting.

However, in this case, the width used is the character width of the font's digit 0 (ASCII $30).

This feature makes it easier to line up columns of figures. It makes all digits, spaces, periods, and the like, the same width. "Width," here, means character width; that is, the pen displacement after the character is drawn. (The image width of the characters remains unchanged.) Of course, standard nonproportionality would also make everything line up, but in most fonts, *widMax* is a good deal more than the width of a digit, which causes numbers to end up spaced too far apart.

Because the width used in numeric spacing is usually less than *widMax,* some characters—for instance, W's and M's—end up overlapping other characters. Consequently, numeric spacing is useful with the characters most commonly used with numbers—space, period, and so on—but is not appropriate for general text.

If you want absolute control over the width of characters, you can use numeric spacing or the standard nonproportionality and then adjust it to your tastes using the SetCharExtra routine.

When numeric spacing is in effect, *chExtra, spExtra,* style modifications, and the like, are applied to the character width after it has been set to the width of the digit 0.

Bits 1 and 0 should not both be set to 1.

Bit 2: Bit 2 controls how the foreground and background colors in the GrafPort are applied to text when it is drawn. If bit 2 is 0, the foreground and background colors are treated as pixel values (two- or four-bit numbers depending upon the GrafPort's SCB), with all other bits in the word ignored. Each foreground pixel is given the value of the foreground color value, and each background pixel is given the value of the background color value. For example, in 640 mode with a foreground color word of 0110011001100110 and bit 2 set to 0, each pixel will have a value of 10.

If bit 2 is set to 1, the foreground and background colors are treated as a word's worth of pixel values. This feature is useful when you are trying to draw text in 640 mode using dithered colors. Each foreground pixel in a destination word is given the value of the corresponding pixel in the foreground color word. Each background pixel in a destination word is given the value of the corresponding pixel in the background color word. For example, in 640 mode with a foreground color word of 0110011001100110 and bit 2 set to 1, odd-numbered pixels will have a value of 10 and even-numbered pixels will have a value of 01.

Using the QuickDraw II font calls

Text drawing calls

DrawChar, DrawText, DrawString, DrawCString: These calls are used when the specified character or string of characters is drawn, using all the current information—font, style, mode, and so forth. The current pen position is used as the character origin of the first character. The pen is advanced by the sum of the character widths. Note that, although the text image is clipped to the current clip region, the pen is not clipped in any way; the new pen position can be outside the current GrafPort bounds.

Warning

Near the edges of its drawing space (±16K,±16K), QuickDraw II is unreliable; this applies to text drawing as well as to any other kind. Calls that would draw outside the space can cause catastrophic results.

Text width calls

CharWidth, TextWidth, StringWidth, CStringWidth: These calls return the total pen displacement that would result if the character or sequence of characters were to be drawn. Nothing is actually drawn, however. The width calls take into account current styles, *chExtra, spExtra,* font flags, and the like. But they do not take kerning (which is independent of pen displacement) into account; that's a job for the text bounds calls. Note that the width calls only return a pen displacement, not a new pen location. They make no use of the current pen location, and they don't change it.

Text bounds calls

CharBounds, TextBounds, StringBounds, CStringBounds: These calls return the smallest rectangle that would enclose all the foreground and background pixels of the character or string (or text block or C string) of characters if they were to be drawn, starting at the current pen location. The rectangle is given in the local coordinates of the current GrafPort.

Unlike the text width calls, these calls take kerning, as well as pen movement, into account. The bounds rectangle extends to the left as far as the starting pen position or the leftmost kerning pixel (if any) of the text image, whichever is farther to the left. Similarly, it extends as far right as the new pen position or the rightmost kerning pixel, whichever is farther to the right. But at the least, the bounds rectangle is reliable; any pixel that might be changed by a text-drawing call is inside the corresponding bounds rectangle.

The rectangle extends up (from the current pen location) to the ascent line and down to the descent line. It is not clipped to any clipping region. It takes into account style modifications, *chExtra, spExtra,* and so forth. Note that the bounds rectangle is not actually drawn by these calls; its coordinates are simply returned to the application.

Some strings (or text or C strings), or possibly even some characters, may have no foreground or background pixels. Such a character would have to have 0 image width and 0 character width—a space with no length. A string may have 0 length (no characters) or be composed entirely of the spaceless spaces just described. In these cases, the text bounds calls return a degenerate rectangle; that is, one whose right and left edges are the same (namely, the current pen location's X coordinate). The upper and lower edges of the rectangle will be the ascent and descent lines (relative to the pen's Y coordinate), as usual.

Text buffer management calls

SetBufDims, ForceBufDims, SaveBufDims, RestoreBufDims: These calls affect the size of the text buffer and the way it is used.

❖ *Note:* If you are using the Font Manager, it takes care of all this text buffer management for you.

Important

These calls affect the QuickDraw II clip buffer as well as the text buffer!

When a string (or text block or C string) is to be drawn into a pixel image, it is first drawn into the text buffer. Characters of the string that fall far outside the destination's left or right boundaries are not actually drawn into the text buffer; only their character widths are used—to determine where the string actually enters the destination (on the right) and/or what the final pen location should be (on the left).

For the text-drawing calls to handle this safely and efficiently, QuickDraw II must have certain information about the largest pixel image sizes and character sizes it will have to deal with. For one thing, the text buffer must be at least as wide (in pixels) as the widest destination pixel image that may be used (actually, it must be a little wider to avoid disaster when drawing characters that fall partly in and partly out of the destination), and it has to be as high as the highest font. For another thing, to decide if a pen location is so far outside the destination that a character drawn with that origin can't possibly impinge on the destination, QuickDraw II needs to know the width of the widest possible character. "Widest," here, includes not only image width and character width, but also any elongations due to *chExtra, spExtra,* style modifications, and so forth. Any pixel that can be touched by a character's foreground or background must be considered.

This is what *fbrExtent* was created for. It describes how far away from the current pen location any pixel that can be altered can be. But *fbrExtent* depends only on the font and does not take into account style modifications and the like. This is why we have two calls: SetBufDims, which provides generous defaults for any character elongations, and ForceBufDims, which puts things more under the application's control.

When QDStartUp is called, it creates a text buffer that is twice as high as the system font, wide enough to support the *maxWidth* parameter of QDStartUp, and capable of handling characters twice as wide as the system font characters ("wide" in the sense of *fbrExtent*). It also permits the use, with any font, of any *chExtra* ≤ *fbrExtent* (of that font); *spExtra* ≤ *fbrExtent;* and it allocates up to 36 extra pixels per character to accomodate style modifications (bolding, for example, adds 1 pixel to a character, and italicizing a large font can stretch its horizontal extent significantly). If your application is going to deal only with fonts and text display parameters that fall within those limits, you can trust to the defaults and never call SetBufDims or ForceBufDims.

The SetBufDims routine takes three parameters:

maxWidth	INTEGER
maxFontHeight	INTEGER
maxFBRExtent	INTEGER

The *maxWidth* value is the width in bytes (not pixels) of the largest pixel image the application will draw into (a value of 0 indicates screen width). It will override the value supplied to QDStartUp. The *maxFontHeight* value is the height, in pixels, of the tallest font the application will have to work with. The *maxFBRExtent* value is the *fbrExtent* of the widest (that is, greatest *fbrExtent*) font the application will work with. The call resizes the clip buffer and the text buffer to accommodate these sizes.

In addition, SetBufDims pads the text buffer to allow for (1) values of *chExtra* and *spExtra* ≤ the *fbrExtent* of the font in use at any given time and (2) an extra 36 pixels of style modification added to the width of any character.

SetBufDims's three parameters are used to size QuickDraw II's internal buffers. When it's time to actually draw a string, and QuickDraw II must decide which characters might make it into the destination, it uses the *fbrExtent* of the current font (which may be way smaller than *maxFBRExtent*), the current values of *spExtra, chExtra, txFace,* and so on, and for a destination pixel image width, the width of the active portion of the current GrafPort's pixel image (its *minRect,* to be specific). Therefore, large values for SetBufDims's parameters may soak up some memory for the text buffer size but will not cost much in time lost drawing into the text buffer characters that will never make it into the destination. This also means that, once the text buffer is sized, the *maxFBRExtent* parameter can be forgotten. (This is not true for ForceBufDims.)

ForceBufDims takes the same parameters as SetBufDims and performs the same functions; however, it does not pad the text buffer at all. Any extra pixels that might be added to a character bounds width due to *chExtra, spExtra,* style modifications, or whatever, should be added into the *maxFBRExtent* parameter by the application making the call.

ForceBufDims, like SetBufDims, sizes the buffer(s) on the basis of its parameters, and when a string is actually drawn, only the width of the current GrafPort's pixel image is considered, not all of *maxWidth*. But, unlike SetBufDims, ForceBufDims forces QuickDraw II to use the *maxFBRExtent* parameter to decide which characters are in and which out, rather than trying to calculate a current *fbrExtent* value. ForceBufDims is for those times when you're going to do something so unusual that QuickDraw II won't be able to anticipate your actions (such as using very large *chExtra* or *spExtra* values). Consequently, when ForceBufDims is called, its *maxFBRExtent* value must be remembered for subsequent drawing calls. In the SaveBufDims and RestoreBufDims, there is an asymmetry in the parameters handed back, depending on whether the text buffer was originally set (*maxFBRExtent* no longer needed) or forced (*maxFBRExtent* must be remembered). Precise calculations of *maxFBRExtent* for the ForceBufDims call are not necessary; upper limits will do.

You can of course call SetBufDims or ForceBufDims every time you change fonts or even every time you call SetCharExtra and SetTextFace. This is not recommended, however, because sizing (and clearing) buffers can be quite time-consuming. The routines should probably be called only once (if at all), with the maximum realistic values for each of the parameters, and never again.

InflateTextBuffer takes two parameters: *newWidth,* a font width (that is, *fbrExtent* in pixels); and *newHeight,* a font height (in pixels). It then calculates whether the current dimensions of the text buffer are large enough to accommodate a font with that width and height and, if they are not, the routine enlarges the text buffer so that it can handle fonts of that size. The routine will never shrink the size of the text buffer. InflateTextBuffer always pads the value of *newWidth* to allow for style modifications and for reasonable values of *chExtra* and *spExtra.*

❖ *Note:* If the current text buffer size was set by a SetBufDims call, then when InflateTextBuffer enlarges the text buffer, it makes an internal call to SetBufDims, so the new width is also considered a "set" value. However, if the text buffer currently has a "forced" width, as set by ForceBufDims, InflateTextBuffer will enlarge the buffer by first padding the newWidth value and then calling ForceBufDims with this padded value as the "forced" font width.

SaveBufDims and RestoreBufDims are included for orderly context-switching between subprograms. SaveBufDims saves the state of the clip buffer and text buffer sizes in the form of an eight-byte record:

maxWidth	INTEGER
textBufHeight	INTEGER
textBufferWords	INTEGER
fontWidth	INTEGER

The *maxWidth* value is the current value of the application-set maximum pixel image width in bytes. The *textBufHeight* value is the current text buffer height in pixels, and *textBufferWords* is the current width of the text buffer in words. The *fontWidth* value serves two purposes: if it is 0, it means the buffer was set up with a call to SetBufDims; if it is nonzero, the buffer was set up with a call to ForceBufDims, and the value of *fontWidth* is equal to the *maxFBRExtent* parameter used in that call.

RestoreBufDims restores the buffer dimensions on the basis of the record it is given.

Regardless of which call—QDStartUp, SetBufDims, ForceBufDims, or RestoreBufDims—sizes or resizes the text buffer, the application is not responsible for clearing it. The calls take care of clearing the text buffer automatically. Also note that SaveBufDims and RestoreBufDims do not save and restore the contents of the text buffer; they restore only the parameters related to its size.

Font information calls

GetFontInfo, GetFontGlobals, GetFGSize, GetFontLore: These calls are included for gathering information on the current font. GetFontInfo returns information in the following font info record:

ascent	INTEGER
descent	INTEGER
widMax	INTEGER
leading	INTEGER

These values have been modified, if necessary, to reflect style modifications currently in effect.

GetFontGlobals returns a variable-length font globals record as follows:

fgFontID	INTEGER
fgStyle	INTEGER
fgSize	INTEGER
fgVersion	INTEGER
fgWidMax	INTEGER
fgFbrExtent	INTEGER

(Additional fields may be present at the end of the record.)

The *fgWidMax* value is taken from the embedded Macintosh font; all the others are from the Apple IIGS header. They are taken directly from the font and are not modified, regardless of any style modifications in effect.

In the future, more information will probably be added to the font globals record. The GetFGSize routine exists to warn the application about the added information. The routine returns the length in bytes of the font globals record. Future versions of QuickDraw II may add extra information at the ends of their font globals records, but for compatibility, those versions will maintain the documented fields and ordering of earlier versions.

GetFontLore returns the same information as GetFontGlobals in the same order. However, the application can use GetFontLore and specify the maximum number of bytes the application will accept; GetFontLore returns no more than that many bytes in the form of an initial segment of the font globals record. Thus, the application can avoid calling the Memory Manager to size a buffer on the basis of GetFGSize; instead, it can just receive the font globals information that the application knows how to handle. The GetFontLore routine is recommended; GetFontGlobals is maintained for compatibility reasons.

$0104 QDBootInit

Initializes QuickDraw II; called only by the Tool Locator.

Warning

An application must never make this call.

This routine puts the address of the cursor update routine into the bank $E1 vectors.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C Call must not be made by an application.

$0204 QDStartUp

Starts up QuickDraw II for use by an application.

Important

Your application must make this call before it makes any other QuickDraw II calls.

The routine sets the current port to the standard port and clears the screen.

QuickDraw II uses three consecutive pages of bank zero for its direct page space, starting at *dPageAddr*. The *maxWidth* parameter specifies the size in bytes of the largest pixel map that will be drawn to (a value of 0 indicates screen width). Knowing this *maxWidth* allows QuickDraw II to allocate certain buffers only once and keep them throughout the life of the application.

Parameters

Stack before call

previous contents	
dPageAddr	**Word**—Bank $0 starting address for 3 pages of direct-page space
masterSCB	**Word**—Master SCB for Super Hi-Res graphics
maxWidth	**Word**—INTEGER; size in bytes of largest pixel map, 0 for screen width
userID	**Word**—User ID of application
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$0401	`alreadyInitialized`	Attempt made to start up QuickDraw II a second time without first shutting it down
$0410	`screenReserved`	Memory Manager reported screen memory (bank $E1 from $2000 to $9FFF) is already owned by someone else
Memory Manager errors		Returned unchanged

C

```
extern pascal void QDStartUp(dPageAddr,masterSCB,maxWidth,userID)
Word     dPageAddr;
Word     masterSCB;
Word     maxWidth;
Word     userID;
```

$0304 QDShutDown

Shuts down QuickDraw II when an application quits.

Important
If your application has started up QuickDraw II, the application must make this call before it quits.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

Memory Manager errors Returned unchanged

C

```
extern pascal void QDShutDown()
```

$0404 QDVersion

Returns the version number of QuickDraw II.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
versionInfo	**Word**—Version number of QuickDraw II
	← **SP**

Errors

None

C

```
extern pascal Word QDVersion()
```

$0504 QDReset

Resets QuickDraw II; called only when the system is reset.

Warning

An application must never make this call.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C Call must not be made by an application.

$0604 QDStatus

Indicates whether QuickDraw II is active.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
activeFlag	**Word**—BOOLEAN; TRUE if QuickDraw II active, FALSE if inactive
	← **SP**

Errors None

C

```
extern pascal Boolean QDStatus()
```

$8004 AddPt

Adds two specified points together and leaves the result in the destination point. For example, two source points of (10,20) and (1,2) result in a destination point of (11,22).

Parameters

Stack before call

previous contents	
— *srcPtPtr* —	**Long**—POINTER to POINT
— *destPtPtr* —	**Long**—POINTER to POINT used as source and destination
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void AddPt(srcPtPtr,destPtPtr)
Point *srcPtPtr;
Point *destPtPtr;
```

$AC04 CharBounds

Places the character bounds rectangle of a specified character into a specified buffer.

Parameters

Stack before call

previous contents	
theChar	**Word**—Character to use to set bounds
resultPtr	**Long**—POINTER to space for rectangle
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void CharBounds(theChar,resultPtr)

Word     theChar;

Rect *resultPtr;
```

$A804 CharWidth

Returns the character width, in pixels (pen displacement), of a specified character.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
theChar	**Word**—Character to be measured
	← **SP**

Stack after call

previous contents	
charWidth	**Word**—INTEGER; width of character in pixels
	← **SP**

Errors

None

C

```
extern pascal Integer CharWidth(theChar)

Word     theChar;
```

$1504 ClearScreen

Sets the words in screen memory to a specified value. The value is stuffed into each word of screen memory. The *colorWord* value represents a group of adjacent pixels (4 in 320 mode; 8 in 640 mode). See the section "Drawing in Color" in this chapter. ClearScreen is usually used to clear the screen to a solid color.

Parameters

Stack before call

previous contents	
colorWord	**Word**—Color as offset into current color table
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void ClearScreen(colorWord)

Word     colorWord;
```

$2604 ClipRect

Changes the clipping region of the current GrafPort to a rectangle that is equivalent to a specified rectangle.

Parameters

Stack before call

previous contents	
-- *rectPtr* --	**Long**—POINTER to RECT defining rectangle
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors   Returned unchanged

C

```
extern pascal void ClipRect(rectPtr)

Rect *rectPtr;
```

$C204 ClosePoly

Completes the polygon definition process started with an OpenPoly call.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

$0441	`PolyNotOpen`	No polygon open in current GrafPort
Memory Manager errors		Returned unchanged

C

```
extern pascal void ClosePoly()
```

$1A04 ClosePort

Deallocates the clipping and visible regions in a port. If the application disposes of the memory containing the port without first calling ClosePort, the memory associated with the handles is lost and cannot be reclaimed.

Warning
Never close the current port.

Parameters

Stack before call

previous contents	
-- *portPtr* --	**Long**—POINTER to port
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors Returned unchanged

C

```
extern pascal void ClosePort(portPtr)

GrafPortPtr     portPtr;
```

$6E04 CloseRgn

Completes the region-definition process started by an OpenRgn call. The region must have already been created by a NewRgn call, which supplies *rgnHandle.*

Parameters

Stack before call

previous contents	
-- *rgnHandle* --	**Long**—HANDLE to region being set to a collection of points
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$0431	`rgnNotOpen`	No region open in current GrafPort
Memory Manager errors		Returned unchanged

C

```
extern pascal void CloseRgn(rgnHandle)

RgnHandle     rgnHandle;
```

$6904 CopyRgn

Copies the region definition from one region to another. The *srcRgnHandle* and *destRgnHandle* must have already been created; in particular, this routine does not allocate the *destRgnHandle*.

Parameters

Stack before call

previous contents	
-- *srcRgnHandle* --	**Long**—HANDLE to source region
-- *destRgnHandle* --	**Long**—HANDLE to destination region
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors Returned unchanged

C

```
extern pascal void CopyRgn(srcRgnHandle,destRgnHandle)
RgnHandle     srcRgnHandle;
RgnHandle     destRgnHandle;
```

$AE04 CStringBounds

Places the character bounds rectangle of a specified C string into a specified buffer.

Parameters

Stack before call

previous contents	
cStringPtr	**Long**—POINTER to C string
resultPtr	**Long**—POINTER to space for rectangle
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void CStringBounds(cStringPtr,resultPtr)

Pointer     cStringPtr;

Rect *resultPtr;
```

$AA04 CStringWidth

Returns the sum of all the character widths, in pixels (pen displacements), in a specified C string. This would be the pen displacement if the string were to be drawn.

Parameters

Stack before call

Stack	
previous contents	
wordspace	**Word**—Space for result
-- *cStringPtr* --	**Long**—POINTER to C string
	← **SP**

Stack after call

Stack	
previous contents	
cStringWidth	**Word**—INTEGER; width of C string in pixels
	← **SP**

Errors

None

C

```
extern pascal Integer CStringWidth(cStringPtr)

Pointer     cStringPtr;
```

$7304 DiffRgn

Calculates the difference of the areas enclosed by two regions and places the region definition of the enclosing area in a destination region. The destination region, which may be one of the source regions, must already exist; this routine does not allocate it.

If the *rgn1Handle* is empty, the destination is set to an empty region.

Parameters

Stack before call

Stack	
previous contents	
— *rgn1Handle* —	**Long**—HANDLE to one source region
— *rgn2Handle* —	**Long**—HANDLE to another source region
— *diffRgnHandle* —	**Long**—HANDLE to destination region
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

Memory Manager errors Returned unchanged

C

```
extern pascal void DiffRgn(rgn1Handle,rgn2Handle,diffRgnHandle)
RgnHandle     rgn1Handle;
RgnHandle     rgn2Handle;
RgnHandle     diffRgnHandle;
```

$6804 DisposeRgn

Deallocates the memory for a specified region. For more information about how memory is allocated and deallocated, see Chapter 12, "Memory Manager," in Volume 1.

Parameters

Stack before call

previous contents
-- *rgnHandle* -- — **Long**—HANDLE of region being disposed
← **SP**

Stack after call

previous contents
← **SP**

Errors

Memory Manager errors — Returned unchanged

C

```
extern pascal void DisposeRgn(rgnHandle)

RgnHandle     rgnHandle;
```

$A404 DrawChar

Draws a specified character at the current pen location and updates the pen location.

Parameters

Stack before call

previous contents
theChar

theChar: **Word**—ASCII code of character to be drawn (0–255)

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void DrawChar(theChar)

Word     theChar;
```

$A604 DrawCString

Draws a specified C string at the current pen location and updates the pen location.

Parameters

Stack before call

previous contents
-- *cStringPtr* --

Long—POINTER to C string

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void DrawCString(cStringPtr)

Pointer     cStringPtr;
```

$A504 DrawString

Draws a specified Pascal-type string at the current pen location and updates the pen location.

Parameters

Stack before call

previous contents
-- *stringPtr* --

Long—POINTER to Pascal-type string to be drawn

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void DrawString(stringPtr)

Pointer     stringPtr;
```

$A704 DrawText

Draws specified text at the current pen location and updates the pen location.

Parameters

Stack before call

previous contents	
-- *textPtr* --	**Long**—POINTER to text to be drawn
textLength	**Word**—INTEGER; number of bytes in text to be drawn
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void DrawText(textPtr,textLength)

Pointer     textPtr;

Word     textLength;
```

$7804 EmptyRgn

Indicates whether a specified region is empty.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
— *rgnHandle* —	**Long**—HANDLE to region
	← **SP**

Stack after call

previous contents	
emptyFlag	**Word**—BOOLEAN; TRUE if region is empty, FALSE if not
	← **SP**

Errors

None

C

```
extern pascal Boolean EmptyRgn(rgnHandle)

RgnHandle     rgnHandle;
```

$8304 EqualPt

Indicates whether two points are equal (two equal points have the same Y and X coordinates).

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *point1Ptr* --	**Long**—POINTER to first POINT
-- *point2Ptr* --	**Long**—POINTER to second POINT
	← **SP**

Stack after call

previous contents	
equalFlag	**Word**—BOOLEAN; TRUE if points are equal, FALSE if not
	← **SP**

Errors

None

C

```
extern pascal Boolean EqualPt(point1Ptr,point2Ptr)

Point *point1Ptr;

Point *point2Ptr;
```

$5104 EqualRect

Indicates whether two rectangles are equal. The two rectangles must have identical sizes and locations to be considered equal. Any two empty rectangles are always equal.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *rect1Ptr* --	**Long**—POINTER to RECT defining one rectangle
-- *rect2Ptr* --	**Long**—POINTER to RECT defining other rectangle
	← **SP**

Stack after call

previous contents	
equalFlag	**Word**—BOOLEAN; TRUE if rectangles are equal, FALSE if not
	← **SP**

Errors

None

C

```
extern pascal Boolean EqualRect(rect1Ptr,rect2Ptr)

Rect *rect1Ptr;

Rect *rect2Ptr;
```

$7704 EqualRgn

Indicates whether two regions are equal. The two regions must have identical sizes, shapes, and locations to be considered equal. Any two empty regions are always equal.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *rgn1Handle* --	**Long**—HANDLE to one region
-- *rgn2Handle* --	**Long**—HANDLE to other region
	← **SP**

Stack after call

previous contents	
equalFlag	**Word**—BOOLEAN; TRUE if regions are equal, FALSE if not
	← **SP**

Errors

Memory Manager errors — Returned unchanged

C

```
extern pascal Boolean EqualRgn(rgn1Handle,rgn2Handle)

RgnHandle     rgn1Handle;

RgnHandle     rgn2Handle;
```

$6404 EraseArc

Erases the interior of a specified arc by filling it with the background pattern.

Parameters

Stack before call

previous contents	
-- *rectPtr* --	**Long**—POINTER to RECT defining enclosing rectangle
startAngle	**Word**—INTEGER; starting angle in degrees
arcAngle	**Word**—INTEGER; arc angle in degrees
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void EraseArc(rectPtr,startAngle,arcAngle)

Rect *rectPtr;

Integer     startAngle;

Integer     arcAngle;
```

$5A04 EraseOval

Erases the interior of a specified oval by filling it with the background pattern.

Parameters

Stack before call

previous contents
-- *rectPtr* --

rectPtr: **Long**—POINTER to RECT defining enclosing rectangle

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void EraseOval(rectPtr)

Rect *rectPtr;
```

$BE04 ErasePoly

Erases the interior of a specified polygon by filling it with the background pattern. Because polygons are treated differently than other closed shapes, the frame of the polygon (if drawn) is not completely erased. See the section "Polygons" in this chapter for more information.

Important

Because this call allocates and deallocates some temporary memory space, Memory Manager errors can occur.

Parameters

Stack before call

previous contents
-- *polyHandle* -- — **Long**—HANDLE to polygon
← **SP**

Stack after call

previous contents
← **SP**

Errors

Memory Manager errors Returned unchanged

C

```
extern pascal void ErasePoly(polyHandle)

Handle     polyHandle;
```

$5504 EraseRect

Erases the interior of a specified rectangle by filling it with the background pattern.

Parameters

Stack before call

previous contents
-- *rectPtr* --

rectPtr: **Long**—POINTER to RECT defining rectangle

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void EraseRect(rectPtr)

Rect *rectPtr;
```

$7B04 EraseRgn

Erases the interior of a specified region by filling it with the background pattern.

Parameters

Stack before call

previous contents
-- *rgnHandle* --

rgnHandle: **Long**—HANDLE to region

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void EraseRgn(rgnHandle)

RgnHandle     rgnHandle;
```

$5F04 EraseRRect

Erases the interior of a specified round rectangle by filling it with the background pattern.

The corners of the round rectangle are sections of an oval defined by *ovalHeight* and *ovalWidth*. For more information, see Figure 16-12 in the section "Rectangles" in this chapter.

Parameters

Stack before call

previous contents	
-- *rectPtr* --	**Long**—POINTER to RECT defining enclosing rectangle
ovalWidth	**Word**—INTEGER; width, in pixels, of oval defining rounded corners
ovalHeight	**Word**—INTEGER; height, in pixels, of oval defining rounded corners
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void EraseRRect(rectPtr,ovalWidth,ovalHeight)

Rect *rectPtr;

Word     ovalWidth;

Word     ovalHeight;
```

$6604 FillArc

Fills the interior of a specified arc with a specified pen pattern.

Parameters

Stack before call

previous contents	
rectPtr	**Long**—POINTER to RECT defining rectangle
startAngle	**Word**—INTEGER; starting angle in degrees
arcAngle	**Word**—INTEGER; arc angle in degrees
patternPtr	**Long**—POINTER to pattern
← **SP**	

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void FillArc(rectPtr,startAngle,arcAngle,patternPtr)
Rect *rectPtr;
Integer     startAngle;
Integer     arcAngle;
Pattern     patternPtr;
```

$5C04 FillOval

Fills the interior of a specified oval with a specified pen pattern.

Parameters

Stack before call

previous contents	
-- *rectPtr* --	**Long**—POINTER to RECT defining enclosing rectangle
-- *patternPtr* --	**Long**—POINTER to pattern
← **SP**	

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void FillOval(rectPtr,patternPtr)
Rect *rectPtr;
Pattern     patternPtr;
```

$C004 FillPoly

Fills the interior of a specified polygon with a specified pen pattern. Because polygons are treated differently than other closed shapes, the frame of the polygon (if drawn) is not completely filled. See the section "Polygons" in this chapter for more information.

Important

Because this call allocates and deallocates some temporary memory space, Memory Manager errors can occur.

Parameters

Stack before call

previous contents	
-- *polyHandle* --	**Long**—HANDLE to polygon
-- *patternPtr* --	**Long**—POINTER to pattern
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors Returned unchanged

C

```
extern pascal void FillPoly(polyHandle,patternPtr)

Handle     polyHandle;

Pattern     patternPtr;
```

$5704 FillRect

Fills the interior of a specified rectangle with a specified pen pattern.

Parameters

Stack before call

previous contents

rectPtr — **Long**—POINTER to rectangle

patternPtr — **Long**—POINTER to pattern

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

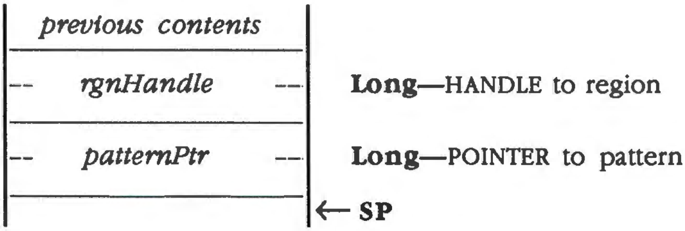
```
extern pascal void FillRect(rectPtr,patternPtr)
Rect *rectPtr;
Pattern     patternPtr;
```

$7D04 FillRgn

Fills the interior of a specified region with a specified pen pattern.

Parameters

Stack before call



Stack after call

Errors

None

C

```
extern pascal void FillRgn(rgnHandle,patternPtr)

RgnHandle     rgnHandle;

Pattern     patternPtr;
```

$6104 FillRRect

Fills the interior of a specified round rectangle with a specified pen pattern.

The corners of the round rectangle are sections of an oval defined by *ovalHeight* and *ovalWidth*. For more information, see Figure 16-12 in the section "Rectangles" in this chapter.

Parameters

Stack before call

previous contents	
-- *rectPtr* --	**Long**—POINTER to RECT defining enclosing rectangle
ovalWidth	**Word**—INTEGER; width, in pixels, of oval defining rounded corners
ovalHeight	**Word**—INTEGER; height, in pixels, of oval defining rounded corners
-- *patternPtr* --	**Long**—POINTER to pattern
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void FillRRect(rectPtr,ovalWidth,ovalHeight,patternPtr)

Rect *rectPtr;

Word     ovalWidth;

Word     ovalHeight;

Pattern     patternPtr;
```

$CC04 ForceBufDims

Sets the size of the QuickDraw II clipping and text buffers, but does not pad *maxFBRExtent* in any way. The *maxFBRExtent* value must include not only the greatest width of the character but also any extra width necessary because of style modifications, *chExtra,* and *spExtra*.

❖ *Note:* You need to make this call only if your application is going to use, or allow the user to choose, fonts that have unusually large values of *chExtra* and *spExtra*. See the section "Fonts and Text in QuickDraw II" in this chapter for more information.

Although SetBufDims and ForceBufDims may be called at any time, it is usually best to call them once with reasonable maximum values early in the application (if at all), because claiming and clearing a buffer can take lots of time.

Parameters

Stack before call

previous contents	
maxWidth	**Word**—INTEGER; width, in bytes, of widest pixel map to be used
maxFontHeight	**Word**—INTEGER; height, in pixels, of tallest font the application will use
maxFBRExtent	**Word**—INTEGER; greatest *fbrExtent* in pixels of any font to be used
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors — Returned unchanged

C

```
extern pascal void ForceBufDims(maxWidth,maxFontHeight,maxFBRExtent)
Word     maxWidth;
Word     maxFontHeight;
Word     maxFBRExtent;
```

$6204 FrameArc

Draws the frame of a specified arc using the current pen mode, pen pattern, and pen size. Only pixels entirely within the rectangle are affected.

Parameters

Stack before call

previous contents	
— *rectPtr* —	**Long**—POINTER to RECT defining enclosing rectangle
startAngle	**Word**—INTEGER; starting angle in degrees
arcAngle	**Word**—INTEGER; arc angle in degrees
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void FrameArc(rectPtr,startAngle,arcAngle)

Rect *rectPtr;

Integer     startAngle;

Integer     arcAngle;
```

$5804 FrameOval

Draws the frame of a specified oval using the current pen mode, pen pattern, and pen size. Only pixels entirely within the rectangle are affected.

Important

If a region is open, this command contributes to the region definition; this can cause Memory Manager errors to occur.

Parameters

Stack before call

previous contents	
-- *rectPtr* --	**Long**—POINTER to RECT defining enclosing rectangle
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors   Returned unchanged

C

```
extern pascal void FrameOval(rectPtr)

Rect *rectPtr;
```

$BC04 FramePoly

Draws the frame of a specified polygon using the current pen mode, pen pattern, and pen size. The polygon is framed with a series of LineTo calls.

Important

If this call is used, parts of the frame are not affected by ErasePoly and FillPoly calls. You can, for example, erase the frame by using another FramePoly call using the background pattern. In addition, if a region is open, this command contributes to the region definition; this can cause Memory Manager errors to occur.

Parameters

Stack before call

previous contents	
-- *polyHandle* --	**Long**—HANDLE to polygon
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors — Returned unchanged

C

```
extern pascal void FramePoly(polyHandle)

Handle     polyHandle;
```

$5304 FrameRect

Draws the frame of a specified rectangle using the current pen mode, pen pattern, and pen size. Only pixels entirely within the rectangle are affected.

Important

If a region is open, this command contributes to the region definition; this can cause Memory Manager errors to occur.

Parameters

Stack before call

previous contents	
-- *rectPtr* --	**Long**—POINTER to RECT defining rectangle
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors Returned unchanged

C

```
extern pascal void FrameRect(rectPtr)

Rect *rectPtr;
```

$7904 FrameRgn

Draws the frame of a specified region using the current pen mode, pen pattern, and pen size. Only pixels entirely inside the region are affected.

If a region is open and being formed, the outline of the region being framed is added to the open region's boundary.

Important

If a region is open, this command contributes to the region definition; this can cause Memory Manager errors to occur.

Parameters

Stack before call

previous contents	
-- *rgnHandle* --	**Long**—HANDLE to region
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors — Returned unchanged

C

```
extern pascal void FrameRgn(rgnHandle)

RgnHandle     rgnHandle;
```

$5D04 FrameRRect

Draws the frame of a specified round rectangle using the current pen mode, pen pattern, and pen size. Only pixels entirely within the rectangle are affected.

Important

If a region is open, this command contributes to the region definition; this can cause Memory Manager errors to occur.

The corners of the round rectangle are sections of an oval defined by *ovalHeight* and *ovalWidth*. For more information, see Figure 16-12 in the section "Rectangles" in this chapter.

Parameters

Stack before call

previous contents	
rectPtr	**Long**—POINTER to RECT defining enclosing rectangle
ovalWidth	**Word**—INTEGER; width, in pixels, of oval defining rounded corners
ovalHeight	**Word**—INTEGER; height, in pixels, of oval defining rounded corners
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors — Returned unchanged

C

```
extern pascal void FrameRRect(rectPtr,ovalWidth,ovalHeight)

Rect *rectPtr;

Word     ovalWidth;

Word     ovalHeight;
```

$0904 GetAddress

Returns a pointer to a specified table. QuickDraw II contains a number of tables that may be useful. The GetAddress call allows you to access these tables.

The current *tableIDs* are as follows:

$0001	`screenTable`
$0002	`conTable320`
$0003	`conTable640`

Important

If your application is using one of these tables, make sure the application obtains the correct pointer every time it runs. These tables will move as the ROM version changes.

The screen table has 200 two-byte entries. Each entry is the address of the start of a scan line in the display buffer. The zeroth entry is $2000, which is the address of scan line 0; entry 1 is $20A0, which is the address of scan line 1; and so on.

The `conTable320` and `conTable640` tables are used to convert from bytes that are one bit per pixel to bytes that are four and two bits per pixel respectively. The `conTable320` table has 256 four-byte entries; the `conTable640` table has 256 two-byte entries. These entries are the two- and four-bit-per-pixel representations of one-bit-per-pixel bytes.

Parameters

Stack before call

previous contents	
longspace	**Long**—Space for result
tableID	**Word**—INTEGER; ID of table whose pointer will be retrieved
	← **SP**

Stack after call

previous contents	
tablePtr	**Long**—POINTER to table in ROM
	← **SP**

Errors None

C

```
extern pascal Pointer GetAddress(tableID)
Word     tableID;
```

Assembly-language example

The byte containing $37 appears as follows in one-, two-, and four-bit-per-pixel mode:

One-bit	Two-bit	Four-bit
%00110111	%00 00 11 11 00 11 11 11	$00FF 0FFF

The two- and four-bit versions would be obtained from the table as follows:

```
lda OneBit              lda OneBit              ; Pick up the byte
and #$00FF              and #$00FF              ; Mask off the high byte
asl a                   asl a                   ; Multiply by 2 or 4
tay                     asl a
lda [TwoBitTable],y     tay                     ; Put result in y
                        lda [FourBitTable],y    ; Load out of table through y
                        tax                     ; Save in x
                        iny                     ; Bump y
                        iny
                        lda [FourBitTable],y    ; Get the second word
```

In both cases, the addresses obtained from GetAddress are already on the direct page.

$B104 GetArcRot

Returns the value of the *arcRot* field in the current GrafPort.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
arcRot	**Word**—INTEGER; value of *arcRot* field
	← **SP**

Errors

None

C

```
extern pascal Integer GetArcRot()
```

$A304 GetBackColor

Returns the value of the *bgColor* field from the GrafPort.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
backColor	**Word**—INTEGER; value of *bgColor* field
	← **SP**

Errors

None

C

```
extern pascal Word GetBackColor()
```

$3504 GetBackPat

Copies the current background pen pattern from the current GrafPort to a specified location.

Parameters

Stack before call

previous contents
-- *patternPtr* --

patternPtr: **Long**—POINTER to location for pattern

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void GetBackPat(patternPtr)

Pattern     patternPtr;
```

$D504 GetCharExtra

Returns the *chExtra* field from the GrafPort.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
	← **SP**

Stack after call

previous contents	
-- *charExtra* --	**Long**—FIXED; value of *chExtra* field
	← **SP**

Errors

None

C

```
extern pascal Fixed GetCharExtra()
```

$2504 GetClip

Copies the clipping region to a specified region. The destination region must have been created earlier with a NewRgn call.

Parameters

Stack before call

previous contents	
-- *rgnHandle* --	**Long**—HANDLE to region
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors   Returned unchanged

C

```
extern pascal void GetClip(rgnHandle)

RgnHandle     rgnHandle;
```

$C704 GetClipHandle

Returns a copy of the handle to the clipping region.

Parameters

Stack before call

previous contents
-- *longspace* --

Long—Space for result

← **SP**

Stack after call

previous contents
-- *rgnHandle* --

Long—HANDLE to clipping region

← **SP**

Errors

None

C

```
extern pascal RgnHandle GetClipHandle()
```

$1104 GetColorEntry

Returns the value of a specified color in a specified color table.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
tableNumber	**Word**—INTEGER; number of color table
entryNumber	**Word**—INTEGER; number of color to be examined
	← **SP**

Stack after call

previous contents	
color	**Word**—Color of entry
	← **SP**

Errors

$0450	`badTableNum`	Invalid table number; 0 to 15 are valid
$0451	`badColorNum`	Invalid color number; 0 to 15 are valid

C

```
extern pascal Word GetColorEntry(tableNumber,entryNumber)

Word     tableNumber;

Word     entryNumber;
```

$0F04 GetColorTable

Fills a specified color table with the contents of another color table.

Parameters

Stack before call

previous contents	
tableNumber	**Word**—INTEGER; color table to be copied
-- *destTablePtr* --	**Long**—POINTER to color table to receive new values
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$0450 badTableNum Invalid table number; 0 to 15 are valid

C

```
extern pascal void GetColorTable(tableNumber,destTablePtr)
Word        tableNumber;
ColorTable     destTablePtr;
```

$8F04 GetCursorAdr

Returns a pointer to the current cursor record. See the section "Cursors" in this chapter for the definition of the cursor record.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
	← **SP**

Stack after call

previous contents	
-- *cursorPtr* --	**Long**—POINTER to current cursor record
	← **SP**

Errors

None

C

```
extern pascal Pointer GetCursorAdr()
```

$CF04 GetFGSize

Returns the size of the font globals record. The font globals record, which contains information about the font, may increase in length in future versions of QuickDraw II. The GetFGSize routine tells your application how much space to allocate for the record.

This call is primarily intended to provide backward compatibility. Under normal circumstances, you'll probably prefer to use the GetFontLore routine. See the section "GetFontLore" in this chapter.

❖ *Note:* The information in the record will only increase. Fields that have been defined will not disappear, but additional fields may be added at the end of the font globals record.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
fgSize	**Word**—INTEGER; size of font globals record
	← **SP**

Errors

None

C

```
extern pascal Word GetFGSize()
```

$9504 GetFont

Returns a handle to the current font.

Parameters

Stack before call

previous contents
-- *longspace* --

Long—Space for result

← **SP**

Stack after call

previous contents
-- *fontHandle* --

Long—HANDLE to font

← **SP**

Errors

None

C

```
extern pascal FontHndl GetFont()
```

$9904 GetFontFlags

Returns the current font flags word. See Figure 16-37 in the section "SetFontFlags" in this chapter for the possible *fontFlags* values.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
fontFlags	**Word**—INTEGER; current font flags word (see Figure 16-37)
	← **SP**

Errors

None

C

```
extern pascal Word GetFontFlags()
```

$9704 GetFontGlobals

Returns information about the font globals record into a specified buffer. The size of the font globals record is returned by the GetFGSize routine, as described in the section "GetFGSize" in this chapter. The information represents the GrafPort's current font and does not reflect style modifications, *chExtra* or *spExtra* fields, and so on. Future versions of QuickDraw II may add more information at the end of this record, but the current fields and their order will be maintained.

This call is primarily intended to provide backward compatibility. Under normal circumstances, you'll probably prefer to use the GetFontLore routine. See the section "GetFontLore" in this chapter.

Parameters

Stack before call

previous contents	
fgRecPtr	**Long**—POINTER to space for font globals record
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void GetFontGlobals(fgRecPtr)

FontGlobalsRecPtr     fgRecPtr;
```

$D104 GetFontID

Returns the *fontID* field of the GrafPort.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
	← **SP**

Stack after call

previous contents	
-- *fontID* --	**Long**—Value of *fontID* field in GrafPort
	← **SP**

Errors

None

C

```
extern pascal LongWord GetFontID()
```

❖ *Note:* C Pascal-type functions do not deal properly with data structures returned on the stack. The Long result returned by this call can be passed to any calls requiring a font ID as a parameter. You cannot use the C dot operator to access the individual font ID fields within the value returned by this call.

$9604 GetFontInfo

Returns information about the current font in a specified buffer. The information in the *fontInfo* record does reflect current style modifications, but not the values of the *chExtra* and *spExtra* fields of the GrafPort. See the section "Font Information Calls" in this chapter for the definition of the font info record.

Your application can use the information returned in the *fontInfo* record to determine the spacing between lines of text. Normal spacing is ascent plus descent plus leading.

Parameters

Stack before call

previous contents	
-- *fontInfoRecPtr* --	**Long**—POINTER to space for *fontInfo* record
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void GetFontInfo(fontInfoRecPtr)

FontInfoRecPtr     fontInfoRecPtr;
```

$D904 GetFontLore

Returns information, up to a specified number of bytes, about the current font in a specified buffer. The routine returns the same values as the GetFontGlobals call (see the section "Font Information Calls" in this chapter), except that GetFontLore will not return more bytes than are specified in the *recordSize* parameter. Thus, you can set aside a fixed amount of space for the record.

Important

This call is available in Version 2.0 or later of QuickDraw II.

You can specify the number of bytes as equal to the maximum number of bytes in the record for a particular version of QuickDraw II. Future versions of QuickDraw II may add more information at the end of this record, but the current fields and their order will be maintained.

The *numBytesXfer* value may sometimes be less than *recordSize*—for example, if the *recordSize* is larger than the number of bytes that GetFontLore has to return.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
recordPtr	**Long**—POINTER to space for record
recordSize	**Word**—Maximum number of bytes to transfer
	← **SP**

Stack after call

previous contents	
numBytesXfer	**Word**—INTEGER; number of bytes transferred
	← **SP**

Errors

None

C

```
extern pascal Word GetFontLore(recordPtr,recordSize)

FontGlobalsRecPtr     recordPtr;

Word     recordSize;
```

$A104 GetForeColor

Returns the value of the current *fgColor* field from the GrafPort.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
foreColor	**Word**—INTEGER; value of *fgColor* field
	← **SP**

Errors

None

C

```
extern pascal Word GetForeColor()
```

$4504 GetGrafProcs

Returns the pointer to the *grafProcs* record associated with the GrafPort.

Parameters

Stack before call

previous contents
-- *longspace* --
← **SP**

Long—Space for result

Stack after call

previous contents
-- *grafProcsPtr* --
← **SP**

Long—POINTER to *grafProcs* record

Errors

None

C

```
extern pascal QDProcsPtr GetGrafProcs()
```

$1704 GetMasterSCB

Returns a copy of the master SCB.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
masterSCB	**Word**—Master SCB value
	← **SP**

Errors

None

C

```
extern pascal Word GetMasterSCB()
```

$2904 GetPen

Returns the pen location.

❖ *Macintosh programmers:* This routine does not pass the point on the stack; instead, it passes a pointer to the point.

Parameters

Stack before call

previous contents
pointPtr — **Long**—POINTER to POINT
← **SP**

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void GetPen(pointPtr)

Point *pointPtr;
```

$3304 GetPenMask

Returns the pen mask to a specified location.

Parameters

Stack before call

previous contents	
-- *maskPtr* --	**Long**—POINTER to space for mask
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void GetPenMask(maskPtr)
Mask     maskPtr;
```

$2F04 GetPenMode

Returns the pen mode from the current GrafPort. See Table 16-9 in the section "SetPenMode" in this chapter for the pen mode values.

Parameters

Stack before call

Stack	
previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

Stack	
previous contents	
penMode	**Word**—INTEGER; pen mode value, as shown in Table 16-9
	← **SP**

Errors

None

C

```
extern pascal Word GetPenMode()
```

$3104 GetPenPat

Copies the current pen pattern from the current GrafPort to a specified location.

Parameters

Stack after call

previous contents
-- *patternPtr* --

patternPtr: **Long**—POINTER to space for pattern

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void GetPenPat(patternPtr)

Pattern     patternPtr;
```

$2D04 GetPenSize

Returns the current pen size to a specified location.

Parameters

Stack before call

```
|  previous contents  |
|---------------------|
|--    pointPtr    --|   Long—POINTER to space for POINT
|---------------------|
|                     | ← SP
```

Stack after call

```
|  previous contents  |
|---------------------|
|                     | ← SP
```

Errors

None

C

```
extern pascal void GetPenSize(pointPtr)

Point *pointPtr;
```

$2B04 GetPenState

Returns the pen state from the GrafPort to a specified location. See Figure 16-38 in the section "SetPenState" in this chapter for the definition of the pen state record.

Parameters

Stack before call

previous contents
-- *penStatePtr* --
← **SP**

Long—POINTER to space for pen state record

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void GetPenState(penStatePtr)

PenStatePtr     penStatePtr;
```

$3F04 GetPicSave

Returns the value of the *picSave* field of the GrafPort.

Parameters

Stack before call

previous contents
-- *longspace* --

Long—Space for result

← **SP**

Stack after call

previous contents
-- *picSaveValue* --

Long—Current *picSave* value

← **SP**

Errors

None

C

```
extern pascal Longint GetPicSave()
```

$8804 GetPixel

Returns the pixel below and to the right of a specified point.

The *thePixel* result is returned in the lower bits of the word. If the current drawing location has a chunkiness of 2, two bits of the word are valid. If the current drawing location has a chunkiness of 4, four bits of the word are valid.

There is no guarantee that the point actually belongs to the port.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
h	**Word**—Horizontal value of point, in global coordinates
v	**Word**—Vertical value of point, in global coordinates
	← **SP**

Stack after call

previous contents	
thePixel	**Word**—Pixel value
	← **SP**

Errors

None

C

```
extern pascal Word GetPixel(h,v)

Integer     h;

Integer     v;
```

You can also use the following alternate form of the call:

```
extern pascal Word GetPixel(point)

Point     point;
```

$4304 GetPolySave

Returns the value of the *polySave* field of the GrafPort.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
	← **SP**

Stack after call

previous contents	
-- *polySaveValue* --	**Long**—Current *polySave* value
	← **SP**

Errors

None

C

```
extern pascal LongWord GetPolySave()
```

$1C04 GetPort

Returns a pointer to the current GrafPort.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
	← **SP**

Stack after call

previous contents	
-- *resultPtr* --	**Long**—POINTER to GrafPort
	← **SP**

Errors

None

C

```
extern pascal GrafPortPtr GetPort()
```

$1E04 GetPortLoc

Gets the current port's *locInfo* record and puts it at the specified location.

Parameters

Stack before call

previous contents
-- *locInfoPtr* --

Long—POINTER to space for *locInfo* record

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void GetPortLoc(locInfoPtr)

LocInfoPtr     locInfoPtr;
```

$2004 GetPortRect

Returns the current GrafPort's port rectangle.

Parameters

Stack before call

previous contents	
-- *rectPtr* --	**Long**—POINTER to space for RECT defining rectangle
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void GetPortRect(rectPtr)

Rect *rectPtr;
```

$4104 GetRgnSave

Returns the value of the *rgnSave* field of the GrafPort.

Parameters

Stack before call

previous contents	
— *longspace* —	**Long**—Space for result
	← **SP**

Stack after call

previous contents	
— *rgnSaveValue* —	**Long**—Current *rgnSave* value
	← **SP**

Errors

None

C

```
extern pascal LongWord GetRgnSave()
```

$D804 GetRomFont

Fills a specified buffer with information about the font in ROM.

Parameters

Stack before call

previous contents
-- *recordPtr* --
← **SP**

Long—POINTER to space for ROM font record (see Figure 16-33)

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void GetROMFont(recordPtr)

RomFontRecPtr     recordPtr;
```

(continued)

The record

The record pointed to by *recordPtr* has the form illustrated in Figure 16-33.

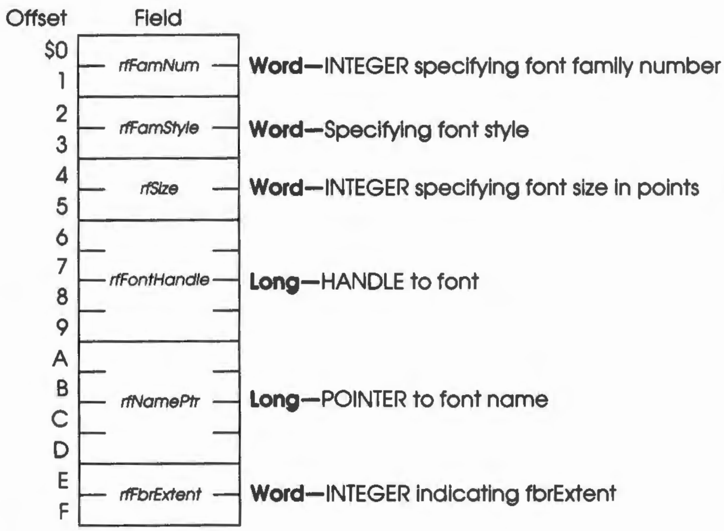


Figure 16-33
ROM font record

$1304 GetSCB

Returns the value of a specified SCB (scan line control byte).

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
scanLine	**Word**—INTEGER; scan line whose SCB is to be examined
	← **SP**

Stack after call

previous contents	
scb	**Word**—Value of SCB
	← **SP**

Errors

$0452	`badScanLine`	Invalid scan line number; 0 to 199 are valid

C

```
extern pascal Word GetSCB(scanLine)

Word     scanLine;
```

$9F04 GetSpaceExtra

Returns the value of the *spExtra* field from the GrafPort.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
	← **SP**

Stack after call

previous contents	
-- *spaceExtra* --	**Long**—FIXED; value of *spExtra* field
	← **SP**

Errors None

C

```
extern pascal Fixed GetSpaceExtra()
```

$0C04 GetStandardSCB

Returns a copy of the standard SCB in the low-order byte of the word.

Parameters

Stack before call

previous contents
wordspace

Word—Space for result

← **SP**

Stack after call

previous contents
scb

Word—Standard SCB (see Figure 16-34)

← **SP**

Errors

None

C

```
extern pascal Word GetStandardSCB()
```

Standard SCB

The standard SCB has values as shown in Figure 16-34.

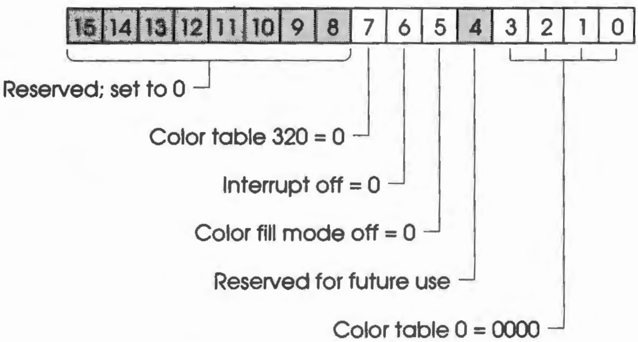


Figure 16-34
Standard SCB

$4904 GetSysField

Returns the value of the *sysField* field of the GrafPort.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
	← **SP**

Stack after call

previous contents	
-- *sysField* --	**Long**—Current *sysField* value
	← **SP**

Errors

None

C

```
extern pascal Longint GetSysField()
```

$B304 GetSysFont

Returns a handle to the current system font.

Parameters

Stack before call

previous contents
-- *longspace* --	**Long**—Space for result
← **SP**

Stack after call

previous contents
-- *fontHandle* --	**Long**—HANDLE to current system for
← **SP**

Errors

None

C

```
extern pascal FontHndl GetSysFont()
```

$9B04 GetTextFace

Returns the current text face. See Figure 16-39 in the section "SetTextFace" in this chapter for the bit values for the *textFace* parameter.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
textFace	**Word**—INTEGER; text face (see Figure 16-39 in "SetTextFace")
	← **SP**

Errors

None

C

```
extern pascal TextStyle GetTextFace()
```

$9D04 GetTextMode

Returns the current text mode. See Table 16-10 in the section "SetTextMode" in this chapter for the modes used only for text.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
textMode	**Word**—INTEGER; text mode (see Table 16-10 in "SetTextMode")
	← **SP**

Errors

None

C

```
extern pascal Word GetTextMode()
```

$D304 GetTextSize

Returns the current value of the *txSize* field of the GrafPort. This value may not be the same as the point size of the current font; to obtain that value, use the GetFontLore or GetFontGlobals routines.

Parameters

Stack before call

Stack	
previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

Stack	
previous contents	
textSize	**Word**—Current value of *txSize* field
	← **SP**

Errors

None

C

```
extern pascal Integer GetTextSize()
```

$4704 GetUserField

Returns the value of the *userField* field of the GrafPort.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
	← **SP**

Stack after call

previous contents	
-- *userField* --	**Long**—Current *userField* value
	← **SP**

Errors

None

C

```
extern pascal Longint GetUserField()
```

$C904 GetVisHandle

Returns a copy of the handle to the visible region.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
	← **SP**

Stack after call

previous contents	
-- *visRgnHandle* --	**Long**—HANDLE to region
	← **SP**

Errors

None

C

```
extern pascal RgnHandle GetVisHandle()
```

$B504 GetVisRgn

Copies the contents of the visible region into a specified region. The region must have already been created with a NewRgn call.

Parameters

Stack before call

previous contents
-- *rgnHandle* --

Long—HANDLE to region

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void GetVisRgn(rgnHandle)

RgnHandle     rgnHandle;
```

$8504 GlobalToLocal

Converts a point from global coordinates to local coordinates. Global coordinates have 0,0 as the upper left corner of the pixel image. Local coordinates are based on the current boundary rectangle of the GrafPort.

Parameters

Stack before call

previous contents
-- *pointPtr* --
← **SP**

pointPtr: **Long**—POINTER to POINT to be converted

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void GlobalToLocal(pointPtr)

Point *pointPtr;
```

$0B04 GrafOff

Turns off the Super Hi-Res graphics mode. The routine affects only the bit in the New Video register that affects what is displayed. It does not change the linearization bit in the field. See the *Apple IIGS Hardware Reference* for more information.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void GrafOff()
```

$0A04 GrafOn

Turns on the Super Hi-Res graphics mode. The routine affects only the bit in the New Video register that affects what is displayed. It does not change the linearization bit in the field. See the *Apple IIGS Hardware Reference* for more information.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void GrafOn()
```

$9004 HideCursor

Hides the cursor by decrementing the cursor level. A cursor level of 0 indicates the cursor is visible; a cursor level less than 0 indicates the cursor is not visible.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void HideCursor()
```

$2704 HidePen

Decrements the pen level. A non-negative pen level indicates that drawing will occur; a negative pen level indicates that drawing will not occur.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void HidePen()
```

$D704 InflateTextBuffer

Ensures that the text buffer is big enough to handle a font with the specified width and height, increasing it if necessary.

This routine is usually used only by the Font Manager, but you may need it if your application is dealing with fonts without the Font Manager's help.

Parameters

Stack before call

previous contents
newWidth — **Word**—INTEGER; width of font
newHeight — **Word**—INTEGER; height of font
← **SP**

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void InflateTextBuffer(newWidth,newHeight)

Word     newWidth;

Word     newHeight;
```

$0D04 InitColorTable

Returns a copy of the standard color table for the current mode, as shown in Table 16-7.

Parameters

Stack before call

previous contents
-- *tablePtr* --

Long—POINTER to space for standard color table (see Table 16-7)

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void InitColorTable(tablePtr)

ColorTable     tablePtr;
```

Standard color tables

The standard color tables are shown in Table 16-7.

Table 16-7
Standard color tables

Pixel value	Name	Master color	Pixel value	Name	Master color
Entries for 320 mode			**Entries for 640 mode**		
0	Black	000	0	Black	000
1	Dark gray	777	1	Red	F00
2	Brown	841	2	Green	0F0
3	Purple	72C	3	White	FFF
4	Blue	00F	4	Black	000
5	Dark green	080	5	Blue	00F
6	Orange	F70	6	Yellow	FF0
7	Red	D00	7	White	FFF
8	Beige	FA9	8	Black	000
9	Yellow	FF0	9	Red	F00
10	Green	0E0	A	Green	0F0
11	Light blue	4DF	B	White	FFF
12	Lilac	DAF	C	Black	000
13	Periwinkle blue	78F	D	Blue	00F
14	Light gray	CCC	E	Yellow	FF0
15	White	FFF	F	White	FFF

$CA04 InitCursor

Reinitializes the cursor. The cursor is set to the arrow cursor and made visible.

This routine also checks the master SCB and sets the cursor accordingly. Use this routine if you want to change modes in the middle of a program. The steps you take to do this are

1. Hide the cursor if it is not already hidden.
2. Set the master SCB to the mode you want.
3. Set all the SCBs to the master SCB.
4. Set the color table the way you want it.
5. Repaint the screen for the new mode.
6. Call InitCursor.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void InitCursor()
```

$1904 InitPort

Initializes specified memory locations as a standard port.

InitPort, unlike the OpenPort routine, assumes that the region handles are valid and does not allocate new handles. Otherwise, InitPort performs the same functions as OpenPort.

Parameters

Stack before call

previous contents	
-- *portPtr* --	**Long**—POINTER to port
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors Returned unchanged

C

```
extern pascal void InitPort(portPtr)

GrafPortPtr     portPtr;
```

$4C04 InsetRect

Insets a specified rectangle by specified displacements. The value specified as *dH* is added to the left and subtracted from the right; the value specified as *dV* is added to the top and subtracted from the bottom.

Parameters

Stack before call

previous contents	
-- *rectPtr* --	**Long**—POINTER to rectangle
dH	**Word**—INTEGER; horizontal displacement
dV	**Word**—INTEGER; vertical displacement
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void InsetRect(rectPtr,dH,dV)
Rect *rectPtr;
Integer    dH;
Integer    dV;
```

$7004 InsetRgn

Shrinks or expands a specified region. All points on the region boundary are moved inward a distance of *dH* horizontally and *dV* vertically. If *dH* or *dV* is negative, the points are moved outward in that direction. InsetRgn leaves the region centered on the same position but moves the outline. InsetRgn of a rectangular region works just like InsetRect.

Parameters

Stack before call

previous contents	
-- *rgnHandle* --	**Long**—HANDLE to region being inset
dH	**Word**—INTEGER; horizontal displacement
dV	**Word**—INTEGER; vertical displacement
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors Returned unchanged

C

```
extern pascal void InsetRgn(rgnHandle,dH,dV)

RgnHandle     rgnHandle;

Integer      dH;

Integer      dV;
```

$6504 InvertArc

Inverts the pixels in the interior of a specified arc.

Parameters

Stack before call

previous contents	
rectPtr	**Long**—POINTER to RECT specifying enclosing rectangle
startAngle	**Word**—INTEGER; starting angle in degrees
arcAngle	**Word**—INTEGER; arc angle in degrees
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void InvertArc(rectPtr,startAngle,arcAngle)

Rect *rectPtr;

Integer     startAngle;

Integer     arcAngle;
```

$5B04 InvertOval

Inverts the pixels in the interior of a specified oval.

Parameters

Stack before call

previous contents
-- *rectPtr* --

Long—POINTER to RECT specifying enclosing rectangle

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void InvertOval(rectPtr)

Rect *rectPtr;
```

$BF04 InvertPoly

Inverts the pixels in the interior of a specified polygon. The polygon is inverted by opening a region, drawing lines, closing the region, and inverting the region.

Important

Because this call allocates and deallocates some temporary memory space, Memory Manager errors can occur.

Parameters

Stack before call

previous contents
-- *polyHandle* --

Long—HANDLE to polygon

← **SP**

Stack after call

previous contents

← **SP**

Errors

Memory Manager errors   Returned unchanged

C

```
extern pascal void InvertPoly(polyHandle)

Handle     polyHandle;
```

$5604 InvertRect

Inverts the pixels in the interior of a specified rectangle.

Parameters

Stack before call

previous contents	
rectPtr	**Long**—POINTER to RECT specifying rectangle
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void InvertRect(rectPtr)

Rect *rectPtr;
```

$7C04 InvertRgn

Inverts the pixels in the interior of a specified region.

Parameters

Stack before call

previous contents
-- *rgnHandle* --

Long—HANDLE to region

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void InvertRgn(rgnHandle)

RgnHandle     rgnHandle;
```

$6004 InvertRRect

Inverts the pixels in the interior of a specified round rectangle.

The corners of the round rectangle are sections of an oval defined by *ovalHeight* and *ovalWidth*. For more information, see Figure 16-12 in the section "Rectangles" in this chapter.

Parameters

Stack before call

previous contents	
-- *rectPtr* --	**Long**—POINTER to RECT specifying enclosing rectangle
ovalWidth	**Word**—INTEGER; width, in pixels, of oval defining rounded corners
ovalHeight	**Word**—INTEGER; height, in pixels, of oval defining rounded corners
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void InvertRRect(rectPtr,ovalWidth,ovalHeight)

Rect *rectPtr;

Word     ovalWidth;

Word     ovalHeight;
```

$C304 KillPoly

Disposes of a specified polygon.

Parameters

Stack before call

previous contents
-- *polyHandle* --
← **SP**

polyHandle: **Long**—HANDLE to polygon to be killed

Stack after call

previous contents
← **SP**

Errors

Memory Manager errors Returned unchanged

C

```
extern pascal void KillPoly(polyHandle)

Handle     polyHandle;
```

$3D04 Line

Draws a line from the current pen location to a new point specified by the horizontal and vertical displacements.

Important
If a region is open, this command contributes to the region definition; this can cause Memory Manager errors to occur.

Parameters

Stack before call

Stack	
previous contents	
dH	**Word**—INTEGER; horizontal displacement in points
dV	**Word**—INTEGER; vertical displacement in points
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

Memory Manager errors Returned unchanged

C

```
extern pascal void Line(dH,dV)

Integer     dH;

Integer     dV;
```

$3C04 LineTo

Draws a line from the current pen location to a specified point. The point must be expressed in local coordinates.

Important
If a region is open, this command contributes to the region definition; this can cause Memory Manager errors to occur.

Parameters

Stack before call

previous contents	
h	**Word**—INTEGER; horizontal point to which line will be drawn
v	**Word**—INTEGER; vertical point to which line will be drawn
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors — Returned unchanged

C

```
extern pascal void LineTo(h,v)

Integer     h;

Integer     v;
```

You can also use the following alternate form of the call:

```
extern pascal void LineTo(point)

Point     point;
```

$8404 LocalToGlobal

Converts a point from local coordinates to global coordinates. Local coordinates are based on the current boundary rectangle of the GrafPort. Global coordinates have 0,0 as the upper left corner of the pixel image.

Parameters

Stack before call

previous contents
-- *pointPtr* --

pointPtr: **Long**—POINTER to POINT to be converted

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void LocalToGlobal(pointPtr)

Point *pointPtr;
```

$C504 MapPoly

Maps a specified polygon from a source rectangle to a destination rectangle.

Parameters

Stack before call

previous contents	
-- *polyHandle* --	**Long**—HANDLE to polygon to be mapped
-- *srcRectPtr* --	**Long**—POINTER to RECT defining source rectangle
-- *destRectPtr* --	**Long**—POINTER to RECT defining destination rectangle
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void MapPoly(polyHandle,srcRectPtr,destRectPtr)
Handle     polyHandle;
Rect *srcRectPtr;
Rect *destRectPtr;
```

$8A04 MapPt

Maps a specified point from a source rectangle to a destination rectangle.

Parameters

Stack before call

previous contents	
-- *pointPtr* --	**Long**—POINTER to POINT
-- *srcRectPtr* --	**Long**—POINTER to RECT defining source rectangle
-- *destRectPtr* --	**Long**—POINTER to RECT defining destination rectangle
← **SP**	

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void MapPt(pointPtr,srcRectPtr,destRectPtr)

Point *pointPtr;

Rect *srcRectPtr;

Rect *destRectPtr;
```

$8B04 MapRect

Maps a specified rectangle from a source rectangle to a destination rectangle.

Parameters

Stack before call

previous contents	
-- *rectPtr* --	**Long**—POINTER to RECT defining rectangle to be mapped
-- *srcRectPtr* --	**Long**—POINTER to RECT defining source rectangle
-- *destRectPtr* --	**Long**—POINTER to RECT defining destination rectangle
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void MapRect(rectPtr,srcRectPtr,destRectPtr)

Rect *rectPtr;

Rect *srcRectPtr;

Rect *destRectPtr;
```

$8C04 MapRgn

Maps a specified region from a source rectangle to a destination rectangle.

Parameters

Stack before call

previous contents	
-- *mapRgnHandle* --	**Long**—HANDLE to region to be mapped
-- *srcRectPtr* --	**Long**—POINTER to RECT defining source rectangle
-- *destRectPtr* --	**Long**—POINTER to RECT defining destination rectangle
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors Returned unchanged

C

```
extern pascal void MapRgn(mapRgnHandle,srcRectPtr,destdRectPtr)

RgnHandle     mapRgnHandle;

Rect *srcRectPtr;

Rect *destdRectPtr;
```

$3B04 Move

Moves the current pen location by specified horizontal and vertical displacements.

Parameters

Stack before call

previous contents	
dH	**Word**—INTEGER; horizontal displacement
dV	**Word**—INTEGER; vertical displacement
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void Move(dH,dV)

Integer     dH;
Integer     dV;
```

$2204 MovePortTo

Changes the location of the current GrafPort's port rectangle. This routine does not affect the pixel image but changes the active area of the GrafPort. The call is normally used by the Window Manager.

Parameters

Stack before call

previous contents	
h	**Word**—INTEGER; horizontal coordinate of upper left corner
v	**Word**—INTEGER; vertical coordinate of upper left corner
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void MovePortTo(h,v)

Integer     h;

Integer     v;
```

$3A04 MoveTo

Moves the current pen location to a specified point. The point is specified in local coordinates.

Parameters

Stack before call

previous contents
h — **Word**—INTEGER; horizontal point in local coordinates
v — **Word**—INTEGER; vertical point in local coordinates
← **SP**

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void MoveTo(h,v)

Integer     h;

Integer     v;
```

You can also use the following alternate form of the call:

```
extern pascal void MoveTo(point)

Point     point;
```

$6704 NewRgn

Allocates space for a new region and initializes it to an empty region. The empty region for this purpose is a rectangular region with a bounding box of (0,0,0,0).

Important

NewRgn is the only routine that creates a new region; all other routines work with existing regions.

Parameters

Stack before call

previous contents	
— *longspace* —	**Long**—Space for result
	← **SP**

Stack after call

previous contents	
— *rgnHandle* —	**Long**—HANDLE to new region
	← **SP**

Errors

Memory Manager errors Returned unchanged

C

```
extern pascal RgnHandle NewRgn()
```

$5204 NotEmptyRect

Indicates whether a specified rectangle is not empty. An empty rectangle has the top greater than or equal to the bottom or the left greater than or equal to the right.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *rectPtr* --	**Long**—POINTER to RECT defining rectangle
	← **SP**

Stack after call

previous contents	
notEmptyFlag	**Word**—BOOLEAN; TRUE if rectangle not empty, FALSE if empty
	← **SP**

Errors

None

C

```
extern pascal Boolean NotEmptyRect(rectPtr)

Rect *rectPtr;
```

$9204 ObscureCursor

Hides the cursor until the mouse moves. This routine can get the cursor out of the way of typing.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

None

C

```
extern pascal void ObscureCursor()
```

$C404 OffsetPoly

Offsets a specified polygon by specified horizontal and vertical displacements.

Parameters

Stack before call

previous contents	
-- *polyHandle* --	**Long**—HANDLE to polygon
dH	**Word**—INTEGER; horizontal displacement in pixels
dV	**Word**—INTEGER; vertical displacement in pixels
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors   Returned unchanged

C

```
extern pascal void OffsetPoly(polyHandle,dH,dV)

Handle      polyHandle;

Integer      dH;

Integer      dV;
```

$4B04 OffsetRect

Offsets a specified rectangle by specified displacements. The value of *dH* is added to the left and right; the value of *dV* is added to the top and bottom.

Parameters

Stack before call

Stack	
previous contents	
-- *rectPtr* --	**Long**—POINTER to rectangle
dH	**Word**—INTEGER; horizontal displacement in pixels
dV	**Word**—INTEGER; vertical displacement in pixels
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void OffsetRect(rectPtr,dH,dV)
Rect *rectPtr;
Integer     dH;
Integer     dV;
```

$6F04 OffsetRgn

Moves a region on the coordinate plane a distance of *dH* horizontally and *dV* vertically. The region retains its size and shape.

Parameters

Stack before call

previous contents	
-- *rgnHandle* --	**Long**—HANDLE to region being offset
dH	**Word**—INTEGER; horizontal displacement in pixels
dV	**Word**—INTEGER; vertical displacement in pixels
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors Returned unchanged

C

```
extern pascal void OffsetRgn(rgnHandle,dH,dV)

RgnHandle     rgnHandle;

Integer     dH;

Integer     dV;
```

$C104 OpenPoly

Returns a handle to a polygon data structure that will be updated by future LineTo calls.

The polygon is completed by making a ClosePoly call.

Parameters

Stack before call

previous contents
-- *longspace* --

Long—Space for result

← **SP**

Stack after call

previous contents
-- *polyHandle* --

Long—HANDLE to polygon

← **SP**

Errors

$0440	`polyAlreadyOpen`	Polygon already open and being saved in current GrafPort
Memory Manager errors		Returned unchanged

C

```
extern pascal Handle OpenPoly()
```

$1804 OpenPort

Initializes specified memory locations as a standard GrafPort, allocates a new visible region and a new clipping region, and makes the GrafPort the current port.

Parameters

Stack before call

previous contents
-- *portPtr* --

Long—POINTER to space for GrafPort record

← **SP**

Stack after call

previous contents

← **SP**

Errors

Memory Manager errors Returned unchanged

C

```
extern pascal void OpenPort(portPtr)

GrafPortPtr     portPtr;
```

$6D04 OpenRgn

Allocates temporary space and starts saving lines and framed shapes for later processing as a region definition. The routine takes no inputs; instead, it allocates memory to hold information about the region being created. When the CloseRgn routine is called, the region is created and this memory is freed.

While the region is open, all calls to Line, LineTo, FrameRect, FrameOval, FrameRRect, FrameRgn, and FramePoly contribute to the region definition.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

$0430	`rgnAlreadyOpen`	Region already being saved in current GrafPort
Memory Manager errors		Returned unchanged

C

```
extern pascal void OpenRgn()
```

$6304 PaintArc

Paints the interior of a specified arc using the current pen mode and pen pattern.

Parameters

Stack before call

previous contents	
rectPtr	**Long**—POINTER to RECT defining enclosing rectangle
startAngle	**Word**—INTEGER; starting angle in degrees
arcAngle	**Word**—INTEGER; arc angle in degrees
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void PaintArc(rectPtr,startAngle,arcAngle)

Rect *rectPtr;

Integer    startAngle;

Integer    arcAngle;
```

$5904 PaintOval

Paints the interior of a specified oval using the current pen mode and pen pattern.

Parameters

Stack before call

previous contents
-- *rectPtr* --
← **SP**

rectPtr — **Long**—POINTER to RECT defining enclosing rectangle

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void PaintOval(rectPtr)

Rect *rectPtr;
```

$7F04 PaintPixels

Transfers a region of pixels. The pixels are transferred without referencing the current GrafPort. The source and destination are defined in the input, as is the clipping region.

Parameters

Stack before call

previous contents
-- *paintParamPtr* --
← **SP**

paintParamPtr: **Long**—POINTER to parameter block (see Figure 16-35)

Stack after call

previous contents
← **SP**

Errors

$0420	`notEqualChunkiness`	Source and destination pixel images not the same type (one is for 320 mode display and one for 640 mode display)

C

```
extern pascal void PaintPixels(paintParamPtr)

PaintParamPtr     paintParamPtr;
```

Parameter block

The parameter block pointed to by *paintParamPtr* is shown in Figure 16-35.

Offset	Field	
$0–3	*ptrToSourceLocInfo*	**Long**—POINTER to source location information
4–7	*ptrToDestLocInfo*	**Long**—POINTER to destination location information
8–B	*ptrToSourceRect*	**Long**—POINTER to source rectangle
C–F	*ptrToDestPoint*	**Long**—POINTER to destination point
10–11	*mode*	**Word**—mode
12–15	*maskHandle*	**Long**—HANDLE to ClipRgn

Figure 16-35
PaintPixels parameter block

$BD04 PaintPoly

Paints the interior of a specified polygon using the current pen mode and pen pattern.

Important

Because this call allocates and deallocates some temporary memory space, Memory Manager errors can occur.

Parameters

Stack before call

previous contents	
-- *polyHandle* --	**Long**—HANDLE to polygon
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors Returned unchanged

C

```
extern pascal void PaintPoly(polyHandle)

Handle     polyHandle;
```

$5404 PaintRect

Paints the interior of a specified rectangle using the current pen mode and pen pattern.

Parameters

Stack before call

previous contents
-- *rectPtr* --

Long—POINTER to RECT defining enclosing rectangle

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void PaintRect(rectPtr)

Rect *rectPtr;
```

$7A04 PaintRgn

Paints the interior of a specified region using the current pen mode and pen pattern.

Parameters

Stack before call

previous contents	
-- *rgnHandle* --	**Long**—HANDLE to region
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void PaintRgn(rgnHandle)

RgnHandle      rgnHandle;
```

$5E04 PaintRRect

Paints the interior of a specified round rectangle using the current pen mode and pen pattern.

The corners of the round rectangle are sections of an oval defined by *ovalHeight* and *ovalWidth*. For more information, see Figure 16-12 in the section "Rectangles" in this chapter.

Parameters

Stack before call

previous contents	
-- *rectPtr* --	**Long**—POINTER to RECT defining enclosing rectangle
ovalWidth	**Word**—INTEGER; width, in pixels, of oval defining rounded corners
ovalHeight	**Word**—INTEGER; height, in pixels, of oval defining rounded corners
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void PaintRRect(rectPtr,ovalWidth,ovalHeight)

Rect *rectPtr;

Word     ovalWidth;

Word     ovalHeight;
```

$3604 PenNormal

Sets the pen state to the standard state, as shown in Table 16-8. Pen location and visibility are not changed.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void PenNormal()
```

Standard pen state table

Table 16-8
Standard pen state

Pen state	Standard state
PenSize	1,1
PenMode	Copy
PenPat	Black
PenMask	1's

$D604 PPToPort

Transfers pixels from a source pixel map to the current port and clips the pixels to the current visible region and clipping region. The routine differs from PaintPixels in that the current GrafPort is used as the destination.

PPToPort can help you do the clipping correctly when you are painting a pixel image to a window.

Parameters

Stack before call

previous contents	
-- *srcLocPtr* --	**Long**—POINTER to source LocInfo record
-- *srcRectPtr* --	**Long**—POINTER to RECT defining source rectangle
destX	**Word**—INTEGER; X coordinate of upper left corner of destination
destY	**Word**—INTEGER; Y coordinate of upper left corner of destination
transferMode	**Word**—Same as pen mode (see Table 16-9 in "SetPenMode")
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$0420	`notEqualChunkiness`	Source and destination pixel images not the same type (one is for 320 mode display and one for 640 mode display)

C

```
extern pascal void PPToPort(srcLocPtr,srcRectPtr,destX,destY,transferMode)

LocInfoPtr     srcLocPtr;

Rect *srcRectPtr;

Integer     destX;

Integer     destY;

Word     transferMode;
```

You can also use the following alternate form of the call:

```
extern pascal void PPToPort(srcLocPtr,srcRectPtr,dest,transferMode)
LocInfoPtr     srcLocPtr;
Rect *srcRectPtr;
Point     dest;
Word     transferMode;
```

$5004 Pt2Rect

Copies a specified point to the upper left corner of a specified rectangle and another point to the lower right corner of the rectangle.

Parameters

Stack before call

Stack	
previous contents	
-- *point1Ptr* --	**Long**—POINTER to first source POINT
-- *point2Ptr* --	**Long**—POINTER to second source POINT
-- *rectPtr* --	**Long**—POINTER to destination rectangle
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void Pt2Rect(point1Ptr,point2Ptr,rectPtr)
Point *point1Ptr;
Point *point2Ptr;
Rect *rectPtr;
```

$4F04 PtInRect

Detects whether the pixel below and to the right of a specified point is in a specified rectangle. The routine returns TRUE if the pixel is within the rectangle and FALSE if it is not. For example, PtInRect((10,10)),((10,10,20,20)) is TRUE, but PtInRect((20,20)),((10,10,20,20)) is FALSE.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *pointPtr* --	**Long**—POINTER to POINT
-- *rectPtr* --	**Long**—POINTER to RECT defining rectangle
	← **SP**

Stack after call

previous contents	
pointFlag	**Word**—BOOLEAN; TRUE if pixel in rectangle, FALSE if not
	← **SP**

Errors

None

C

```
extern pascal Boolean PtInRect(pointPtr,rectPtr)

Point *pointPtr;

Rect *rectPtr;
```

$7504 PtInRgn

Checks to see whether the pixel below and to the right of a specified point is within a specified region. The routine returns TRUE if the pixel is within the region and FALSE if it is not.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *pointPtr* --	**Long**—POINTER to POINT
-- *rgnHandle* --	**Long**—HANDLE to region
← **SP**	

Stack after call

previous contents	
pixelFlag	**Word**—BOOLEAN; TRUE if pixel is within region, FALSE if not
← **SP**	

Errors

Memory Manager errors   Returned unchanged

C

```
extern pascal Boolean PtInRgn(pointPtr,rgnHandle)

Point *pointPtr;

RgnHandle     rgnHandle;
```

$8604 Random

Returns a pseudorandom number in the range –32768 to 32767. The sequence of numbers generated by repeated calls to this routine depends on the *randomSeed* value set by a SetRandSeed call. In particular, a call to SetRandSeed with a given *randomSeed* value, followed by a sequence of calls to Random (with no SetRandSeed calls in between), will always produce the same sequence of pseudorandom numbers. This can be useful in debugging.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
randomInt	**Word**—INTEGER; pseudorandom number
	← **SP**

Errors

None

C

```
extern pascal Integer Random()
```

$7604 RectInRgn

Checks whether a specified rectangle intersects a specified region.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *rectPtr* --	**Long**—POINTER to RECT defining rectangle
-- *rgnHandle* --	**Long**—HANDLE to region
	← **SP**

Stack after call

previous contents	
encloseFlag	**Word**—BOOLEAN; TRUE if intersection encloses at least a pixel, FALSE if not
	← **SP**

Errors

Memory Manager errors — Returned unchanged

C

```
extern pascal Boolean RectInRgn(rectPtr,rgnHandle)

Rect *rectPtr;

RgnHandle     rgnHandle;
```

$6C04 RectRgn

Destroys previous region information by setting a specified region to a specified rectangle. If the input does not describe a valid rectangle, the region is set to an empty region. If the original region was not rectangular, the region is resized.

Parameters

Stack before call

previous contents	
-- *rgnHandle* --	**Long**—HANDLE to region being set
-- *rectPtr* --	**Long**—POINTER to RECT defining rectangle used as source
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors Returned unchanged

C

```
extern pascal void RectRgn(rgnHandle,rectPtr)

RgnHandle     rgnHandle;

Rect *rectPtr;
```

$CE04 RestoreBufDims

Restores QuickDraw II's internal buffers to the sizes described in the eight-byte record created by the SaveBufDims routine. You can use this routine when you want your application to change temporarily (but be able to restore) the size of the QuickDraw II buffers.

Parameters

Stack before call

previous contents	
— *sizeInfoPtr* —	**Long**—POINTER to record to be restored
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors — Returned unchanged

C

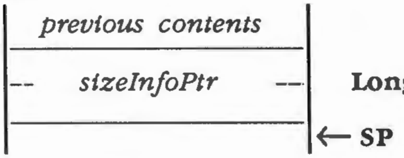
```
extern pascal void RestoreBufDims(sizeInfoPtr)

BufDimRecPtr      sizeInfoPtr;
```

$CD04 SaveBufDims

Saves QuickDraw II's buffer-sizing information in an eight-byte record. You can use this routine when you want your application to change temporarily (but be able to restore) the size of the QuickDraw II buffers. The buffer-sizing record is shown in Figure 16-36.

Parameters

Stack before call



Long—POINTER to space for record (see Figure 16-36)

Stack after call

Errors

None

C

```
extern pascal void SaveBufDims(sizeInfoPtr)
BufDimRecPtr     sizeInfoPtr;
```

Buffer-sizing record

The eight-byte record created by SaveBufDims is shown in Figure 16-36.

Offset	Field	
$0–1	*maxWidth*	**Word**—Application-defined maximum pixel image width
2–3	*textBufHeight*	**Word**—Current text buffer height, in pixels
4–5	*textBufferWords*	**Word**—Current text buffer width, in words
6–7	*fontWidth*	**Word**—Equal to maxFBR Extent

Figure 16-36
BufDimRec

$8904 ScalePt

Scales a specified point from a source rectangle to a destination rectangle.

Parameters

Stack before call

previous contents	
-- *pointPtr* --	**Long**—POINTER to POINT to be scaled
-- *srcRectPtr* --	**Long**—POINTER to RECT defining source rectangle
-- *destRectPtr* --	**Long**—POINTER to RECT defining destination rectangle
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void ScalePt(pointPtr,srcRectPtr,destRectPtr)
Point *pointPtr;
Rect *srcRectPtr;
Rect *destRectPtr;
```

$7E04 ScrollRect

Shifts the pixels inside the intersection of a specified rectangle, visible region, clipping region, port rectangle, and bounds rectangle. No other pixels are affected. The pixels are shifted a distance of *dH* horizontally and *dV* vertically (those shifted out of the scroll area are lost). Positive directions are to the right and down. The background pattern fills the space created by the scroll. In addition, the region for *updateRgnHandle* is changed to the area filled with the background pattern.

❖ *Note:* The update region must be an existing region; ScrollRect does not create it. If you do not want the update region, you may pass NIL.

Parameters

Stack before call

previous contents	
-- *rectPtr* --	**Long**—POINTER to RECT defining rectangle
dH	**Word**—INTEGER; horizontal distance to scroll in pixels
dV	**Word**—INTEGER; vertical distance to scroll in pixels
--*updateRgnHandle* --	**Long**—HANDLE to region
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors Returned unchanged

C

```
extern pascal void ScrollRect(rectPtr,dH,dV,updateRgnHandle)

Rect *rectPtr;

Integer     dH;

Integer     dV;

RgnHandle     updateRgnHandle;
```

$4D04 SectRect

Calculates the intersection of two rectangles and places the intersection in a destination rectangle. The destination rectangle can be one of the source rectangles. If the result is not empty, the output is TRUE; if the result is empty, the output is FALSE.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *rect1Ptr* --	**Long**—POINTER to RECT defining first source rectangle
-- *rect2Ptr* --	**Long**—POINTER to RECT defining second source rectangle
-- *intersectRectPtr* --	**Long**—POINTER to RECT defining destination rectangle
← **SP**	

Stack after call

previous contents	
notEmptyFlag	**Word**—BOOLEAN; TRUE if rectangle not empty, FALSE if empty
← **SP**	

Errors

None

C

```
extern pascal Boolean SectRect(rect1Ptr,rect2Ptr,intersectRectPtr)

Rect *rect1Ptr;

Rect *rect2Ptr;

Rect *intersectRectPtr;
```

$7104 SectRgn

Calculates the intersection of two regions and places the intersection in a destination region. The destination region, which may be one of the source regions, must already exist; SectRgn does not allocate it.

If the regions do not intersect, or if one of the regions is empty, the destination is set to the empty region.

Parameters

Stack before call

Stack	
previous contents	
-- *rgn1Handle* --	**Long**—HANDLE to one source region
-- *rgn2Handle* --	**Long**—HANDLE to another source region
-- *destRgnHandle* --	**Long**—HANDLE to destination region
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

Memory Manager errors — Returned unchanged

C

```
extern pascal void SectRgn(rgn1Handle,rgn2Handle,destRgnHandle)

RgnHandle     rgn1Handle;

RgnHandle     rgn2Handle;

RgnHandle     destRgnHandle;
```

$1404 SetAllSCBs

Sets all SCBs (scan line control bytes) to a specified value.

Parameters

Stack before call

previous contents
newSCB — **Word**—New value for SCBs
← **SP**

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void SetAllSCBs(newSCB)

Word     newSCB;
```

$B004 SetArcRot

Sets the *arcRot* field in the GrafPort to a specified value.

Parameters

Stack before call

previous contents
arcRotValue — **Word**—INTEGER; new value of *arcRot* field
← **SP**

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void SetArcRot(arcRotValue)
Integer     arcRotValue;
```

$A204 SetBackColor

Sets the *bgColor* (background color) field in the GrafPort to a specified value. Background color has either a two- or a four-bit value, depending on the port SCB. If the port SCB indicates 320 mode, the lower four bits of *backColor* are used. If the port SCB indicates 640 mode, the lower two bits of *backColor* are used.

Parameters

Stack before call

Stack	
previous contents	
backColor	**Word**—INTEGER; background color
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetBackColor(backColor)

Word     backColor;
```

$3404 SetBackPat

Sets the background pattern to a specified pattern.

Parameters

Stack before call

previous contents	
-- *patternPtr* --	**Long**—POINTER to pattern
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetBackPat(patternPtr)

Pattern     patternPtr;
```

$CB04 SetBufDims

Sets the size of the QuickDraw II clipping and text buffers. This routine overrides the *maxWidth* value supplied to QDStartUp and the text buffer defaults set at that time.

❖ *Note:* You only need to make this call if your application is going to use, or allow the user to choose, fonts that have unusually large values of *chExtra* and *spExtra*.

SetBufDims pads the text buffer to permit values of *chExtra* ≤ *fbrExtent* (of the currently active font), *spExtra* ≤ *fbrExtent*, and style modifications that add up to 36 pixels to the bounds width (width of foreground and background) of any character.

When QDStartUp is called, it makes an internal call to SetBufDims with the following values:

maxWidth	As supplied by the application
maxFontHeight	2 * (height of system font)
maxFBRExtent	2 * (*fbrExtent* of system font)

Parameters

Stack before call

previous contents	
maxWidth	**Word**—INTEGER; width, in bytes, of widest pixel image to be used
maxFontHeight	**Word**—INTEGER; height, in pixels, of tallest font application will use
maxFBRExtent	**Word**—INTEGER; greatest *fbrExtent*, in pixels, of any font to be used
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors   Returned unchanged

C

```
extern pascal void SetBufDims(maxWidth,maxFontHeight,maxFBRExtent)

Word     maxWidth;

Word     maxFontHeight;

Word     maxFBRExtent;
```

(continued)

More about parameters

The *maxWidth* parameter is the width, in bytes, of the widest pixel image the application will draw into. The *maxFontHeight* parameter is the height of the tallest font the application will use (that is, *fRectHeight* from the font record, computable as *ascent* plus *descent* from the GetFontInfo call).

The *maxFBRExtent* parameter is the greatest *fbrExtent* value of any font the application will use. A field in the font record, *fbrExtent* is returned by a GetFontLore or GetFontGlobals call. It is defined as the greatest (horizontal) distance, in pixels, from the character origin to the farthest foreground or background pixel of any character in the font. For more information, see the section "Fonts and Text in QuickDraw II" in this chapter.

$D404 SetCharExtra

Sets the *chExtra* field in the GrafPort to the specified value. The *chExtra* field is used to add width to every character in the font that has width. It does not affect 0 width characters. This field is present because some fonts that look fine in one graphics mode need a little extra space between characters in another mode.

If you set a very large value of *chExtra*, you may have to change the size of the QuickDraw II buffer. See the sections "SetBufDims" and "ForceBufDims" in this chapter.

Important

SetCharExtra uses FIXED values. You can use the Integer Math Tool Set routine FixRatio to convert values to FIXED values.

Parameters

Stack before call

Stack	
previous contents	
— *charExtra* —	**Long**—FIXED; value for *chExtra* field
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetCharExtra(charExtra)

Fixed     charExtra;
```

$2404 SetClip

Copies a specified region into the clipping region. The handle to the clipping region is not changed.

Parameters

Stack before call

previous contents	
-- *rgnHandle* --	**Long**—HANDLE to region
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors   Returned unchanged

C

```
extern pascal void SetClip(rgnHandle)

RgnHandle     rgnHandle;
```

$C604 SetClipHandle

Sets the *clipRgn* handle field in the GrafPort to a specified value.

Parameters

Stack before call

previous contents
-- *rgnHandle* --

rgnHandle: **Long**—HANDLE to clipping region

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void SetClipHandle(rgnHandle)

RgnHandle     rgnHandle;
```

$1004 SetColorEntry

Sets the value of a color in a specified color table.

Parameters

Stack before call

Stack	Description
previous contents	
tableNumber	**Word**—INTEGER; number of color table
entryNumber	**Word**—INTEGER; number of color to be changed
newColor	**Word**—INTEGER; master color value for color
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

$0450	`badTableNum`	Invalid table number; 0 to 15 are valid
$0451	`badColorNum`	Invalid color number; 0 to 15 are valid

C

```
extern pascal void SetColorEntry(tableNumber,entryNumber,newColor)

Word      tableNumber;

Word      entryNumber;

ColorValue     newColor;
```

$0E04 SetColorTable

Sets a specified color table to specified values. The 16 color tables are stored starting at $9E00. Each table takes $20 bytes. Each word in the table represents one of 4,096 colors. The high-order nibble of the high-order byte is ignored.

Parameters

Stack before call

previous contents	
tableNumber	**Word**—INTEGER; number of table whose color values will be set
srcTablePtr	**Long**—POINTER to color table
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$0450 `badTableNum` Invalid table number; 0 to 15 are valid

C

```
extern pascal void SetColorTable(tableNumber,srcTablePtr)

Word        tableNumber;

ColorTable      srcTablePtr;
```

$8E04 SetCursor

Sets the cursor to an image passed in a specified cursor record. If the cursor is hidden, it remains hidden and appears in the new form when it becomes visible again. If the cursor is visible, it appears in the new form immediately. See the section "Cursors" in this chapter.

Parameters

Stack before call

previous contents	
-- *cursorPtr* --	**Long**—POINTER to cursor record
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors — Returned unchanged

C

```
extern pascal void SetCursor(cursorPtr)

Pointer     cursorPtr;
```

$6A04 SetEmptyRgn

Destroys previous region information by setting a specified region to an empty region. The empty region for this purpose is a rectangular region with a bounding box of (0,0,0,0). If the original region was not rectangular, the region is resized.

Parameters

Stack before call

previous contents
-- *rgnHandle* --
← **SP**

rgnHandle: **Long**—HANDLE to region being modified

Stack after call

previous contents
← **SP**

Errors

Memory Manager errors — Returned unchanged

C

```
extern pascal void SetEmptyRgn(rgnHandle)

RgnHandle     rgnHandle;
```

$9404 SetFont

Sets the current font to a specified font. The call also zeros out the GrafPort's *fontID* field. After the call, you can set the font ID to anything desired by using the QuickDraw II routine SetFontID (see the section "SetFontID" in this chapter).

Important

Under most circumstances, your application should work with the Font Manager to set the font it needs. Use this call only if your application is handling all font manipulation by itself.

Parameters

Stack before call

previous contents	
-- *newFontHandle* --	**Long**—HANDLE to font
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetFont(newFontHandle)

FontHndl     newFontHandle;
```

$9804 SetFontFlags

Sets the font flags word to a specified value. The font flags word is used to indicate special operations performed on the text. At the time of publication, two flags are defined. See Figure 16-37 for the available values for those flags.

Important

The *chExtra* and *spExtra* values, and changes to character width, are applied after the character widths are fixed by the *fontFlags* setting.

Parameters

Stack before call

previous contents
fontFlags — **Word**—INTEGER; font flags (see Figure 16-37)
← **SP**

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void SetFontFlags(fontFlags)

Word     fontFlags;
```

(continued)

Font flags

The values available for *fontFlags* are shown in Figure 16-37.

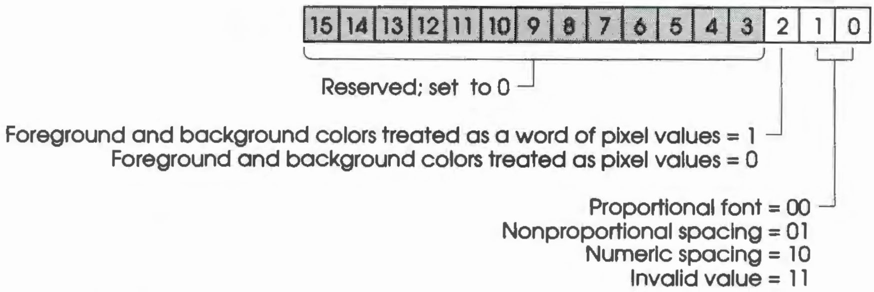


Figure 16-37
Font flags word

If bits 1–0 are set to 00, the font is considered to be a proportionally spaced font.

If bits 1–0 are set to 01, the font is considered to be a fixed-width font (rather than a proportional font), all characters will be equally spaced, and the width of each character will be that of the *widMax* field in the *fontInfo* record.

If bits 1–0 are set to 10, the font is considered to be a fixed-width font, all characters will be equally spaced, and the width of each character will be equal to the character width of the font's 0 (zero) character. This feature makes it easier to line up columns of numbers. Because the width used in numeric spacing is usually less than *widMax*, some characters—for instance, W's and M's—end up overlapping other characters. Consequently, numeric spacing is useful with the characters most commonly used with numbers, such as the space character or the period, but is not appropriate for general text.

If bit 2 is set to 0, the foreground and background colors are treated as pixel values; that is, either as a two- or four-bit number depending on the GrafPort's SCB. The other bits in the word are ignored. Each foreground pixel is given the value of the foreground color value, and each background pixel is given the value of the background color value. For example, in 640 mode with a foreground color word of 0110011001100110 and bit 2 set to 0, each pixel will have a value of 10.

If bit 2 is set to 1, the foreground and background colors are treated as a word's worth of pixel values. This feature is useful when you are trying to draw text in 640 mode using dithered colors. Each foreground pixel in a destination word is given the value of the corresponding pixel in the foreground color word. Each background pixel in a destination word is given the value of the corresponding pixel in the background color word. For example, in 640 mode with a foreground color word of 0110011001100110 and bit 2 set to 1, odd-numbered pixels will have a value of 10 and even-numbered pixels will have a value of 01.

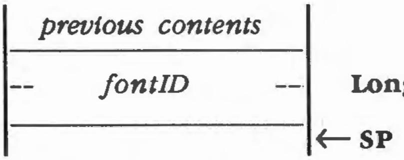
$D004 SetFontID

Sets the *fontID* field in the GrafPort. This routine does not change the current font.

SetFontID is designed for use by the Font Manager for the benefit of the picture routines. The picture routines use the font ID to try to find the font the application really wanted to draw with, rather than the one that was available when the picture was recorded.

Parameters

Stack before call



Stack after call

Errors

None

C

```
extern pascal void SetFontID(fontID)

FontID     fontID;
```

$A004 SetForeColor

Sets the *fgColor* (foreground color) field in the GrafPort to a specified value. Foreground color has either a two- or four-bit value, depending on the port SCB. If the port SCB indicates 320 mode, the lower four bits of *foreColor* are used. If the port SCB indicates 640 mode, the lower two bits of *foreColor* are used.

Parameters

Stack before call

previous contents	
foreColor	**Word**—INTEGER; foreground color
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetForeColor(foreColor)

Word     foreColor;
```

$4404 SetGrafProcs

Sets the *grafProcs* field of the current GrafPort to a specified value.

Parameters

Stack before call

previous contents
— *grafProcsPtr* —

Long—POINTER to *grafProcs* record

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void SetGrafProcs(grafProcsPtr)

QDProcsPtr     grafProcsPtr;
```

$B604 SetIntUse

Indicates to the cursor drawing code whether the code should use scan line interrupts.

QuickDraw II normally uses scan line interrupts to draw the cursor without flicker. If an application wants to use scan line interrupts for some process of its own, it must tell QuickDraw II not to use them.

Parameters

Stack before call

previous contents
useInt

← **SP**

useInt: **Word**—BOOLEAN; TRUE for cursor to use scan line interrupts, FALSE for cursor to not use scan line interrupts

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void SetIntUse(useInt)

Word     useInt;
```

$1604 SetMasterSCB

Sets the master SCB to a specified value. The master SCB is the global mode byte used throughout QuickDraw II. It is used by routines such as InitPort to decide what standard values should be put into the GrafPort.

Parameters

Stack before call

previous contents
masterSCB
← **SP**

masterSCB: **Word**—Value for master SCB (low-order byte only; high-order byte = 0)

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void SetMasterSCB(masterSCB)

Word     masterSCB;
```

$2304 SetOrigin

Adjusts the contents of the port rectangle and the bounds rectangle so the upper left corner of the port rectangle is set to the specified point.

The visible region is also affected, but the clipping region is not. The pen position does not change.

Parameters

Stack before call

previous contents	
h	**Word**—INTEGER; horizontal coordinate of upper left corner
v	**Word**—INTEGER; vertical coordinate of upper left corner
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors Returned unchanged

C

```
extern pascal void SetOrigin(h,v)

Integer    h;

Integer    v;
```

You can also use the following alternate form of the call:

```
extern pascal void SetOrigin(point)

Point    point;
```

$3204 SetPenMask

Sets the pen mask to a specified mask.

Parameters

Stack before call

previous contents
-- *maskPtr* --

maskPtr: **Long**—POINTER to pen mask

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void SetPenMask(maskPtr)

Mask     maskPtr;
```

$2E04 SetPenMode

Sets the current pen mode to a specified pen mode, as shown in Table 16-9.

Parameters

Stack before call

previous contents
penMode — **Word**—INTEGER; new pen mode (see Table 16-9)
← **SP**

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void SetPenMode(penMode)

Word     penMode;
```

Pen modes

Table 16-9 shows the available pen modes. Each 1 and 0 is the value of a bit in a pixel.

❖ *Note:* Special text modes are also available. See Table 16-10 in the section "SetTextMode" in this chapter for those modes.

Table 16-9
Pen modes

Integer	Name	Description
$0000 $8000	modeCopy notCopy	Copy source (or not source) to destination. modeCopy is the typical drawing mode. For text, the fully colored text pixels (both foreground and background) are copied into the destination.

modeCopy		Pen 0	Pen 1
Destination	0	0	1
	1	0	1

notCopy		Pen 0	Pen 1
Destination	0	1	0
	1	1	0

Integer	Name	Description
$0001 $8001	modeOR notOR	Overlay (OR) source (or not source) and destination. Use modeOR to nondestructively overlay new images on top of existing images; use notOR to overlay inverted images. For text, the fully colored text pixels (both foreground and background) are ORed with the destination.

modeOR		Pen 0	Pen 1
Destination	0	0	1
	1	1	1

notOR		Pen 0	Pen 1
Destination	0	1	0
	1	1	1

Integer	Name	Description
$0002 $8002	modeXOR notXOR	Exclusive or (XOR) pen with destination. Use these modes for cursor drawing and rubber-banding. If an image is drawn in XOR mode, the appearance of the destination at the image location can be restored by drawing the image again in XOR mode. For text, the fully colored text pixels (both foreground and background) are XORed with the destination.

modeXOR		Pen 0	Pen 1
Destination	0	0	1
	1	1	0

notXOR		Pen 0	Pen 1
Destination	0	1	0
	1	0	1

Integer	Name	Description
$0003 $8003	modeBIC notBIC	Bit Clear (BIC) pen with destination ((NOT pen) AND destination). Use this mode to explicitly erase (turn off) pixels, often prior to overlaying another image. You can use notBIC to display the intersection of two images. For text, the fully colored text pixels (both foreground and background) are BICed with the destination.

modeBIC		Pen 0	Pen 1
Destination	0	0	0
	1	1	0

notBIC		Pen 0	Pen 1
Destination	0	0	0
	1	0	1

$3004 SetPenPat

Sets the current pen pattern to a specified pen pattern.

Parameters

Stack before call

previous contents
-- *patternPtr* --

Long—POINTER to pattern

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void SetPenPat(patternPtr)

Pattern     patternPtr;
```

$2C04 SetPenSize

Sets the current pen size to a specified pen size.

Parameters

Stack before call

previous contents	
penWidth	**Word**—INTEGER; width of pen in pixels
penHeight	**Word**—INTEGER; height of pen in pixels
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetPenSize(penWidth,penHeight)

Word     penWidth;

Word     penHeight;
```

$2A04 SetPenState

Sets the pen state in the GrafPort to specified values.

Parameters

Stack before call

Stack after call

Errors

None

C

```
extern pascal void SetPenState(penStatePtr)

PenStatePtr     penStatePtr;
```

Pen state record

The record pointed to by *penStatePtr* is formatted as shown in Figure 16-38.

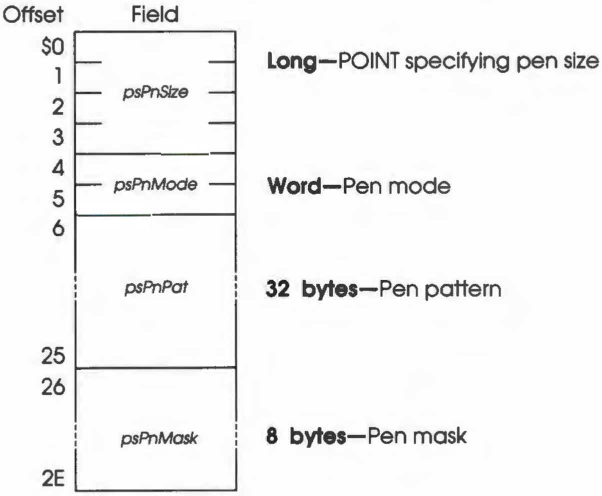


Figure 16-38
Pen state record

$3E04 SetPicSave

Sets the *picSave* field in the GrafPort to a specified value.

Warning
This is an internal routine that should not be used by application programs.

Parameters

Stack before call

previous contents
-- *picSaveValue* --
← SP

picSaveValue: **Long**—New value for *picSave* field

Stack after call

previous contents
← SP

Errors None

C

```
extern pascal void SetPicSave(picSaveValue)

Longint      picSaveValue;
```

$4204 SetPolySave

Sets the *polySave* field in the GrafPort to a specified value.

Warning
This is an internal routine that should not be used by application programs.

Parameters

Stack before call

previous contents
-- *polySaveValue* --

Long—New value for *polySave* field

← **SP**

Stack after call

previous contents

← **SP**

Errors None

C

```
extern pascal void SetPolySave(polySaveValue)

Longint     polySaveValue;
```

$1B04 SetPort

Makes a specified port the current GrafPort.

Parameters

Stack before call

previous contents
portPtr

portPtr: **Long**—POINTER to port

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void SetPort(portPtr)

GrafPortPtr     portPtr;
```

$1D04 SetPortLoc

Sets the current port's *locInfo* record to specified location information.

Parameters

Stack before call

previous contents	
-- *locInfoPtr* --	**Long**—POINTER to location information
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetPortLoc(locInfoPtr)

LocInfoPtr     locInfoPtr;
```

$1F04 SetPortRect

Sets the current GrafPort's port rectangle to the specified rectangle.

Parameters

Stack before call

previous contents
-- *rectPtr* --
← **SP**

rectPtr: **Long**—POINTER to RECT defining rectangle

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void SetPortRect(rectPtr)

Rect *rectPtr;
```

$2104 SetPortSize

Changes the size of the current GrafPort's port rectangle. The routine does not affect the pixel image; it just changes the active area of the GrafPort. Normally, the call is made only by the Window Manager.

Parameters

Stack before call

previous contents	
portWidth	**Word**—INTEGER; width of active area in pixels
portHeight	**Word**—INTEGER; height of active area in pixels
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetPortSize(portWidth,portHeight)

Word     portWidth;

Word     portHeight;
```

$8204 SetPt

Sets a point to specified horizontal and vertical values.

Parameters

Stack before call

previous contents	
— *srcPtPtr* —	**Long**—POINTER to POINT
h	**Word**—INTEGER; horizontal value of point
v	**Word**—INTEGER; vertical value of point
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetPt(srcPtPtr,h,v)
Point *srcPtPtr;
Integer     h;
Integer     v;
```

$8704 SetRandSeed

Sets the seed value for the random number generator. The algorithm uses a 32-bit seed to produce a 16-bit random number. See the section "Random" in this chapter.

Parameters

Stack before call

previous contents
randomSeed
← **SP**

randomSeed: **Long**—LONGINT; seed value

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void SetRandSeed(randomSeed)

Longint     randomSeed;
```

$4A04 SetRect

Sets a specified rectangle to specified values.

Parameters

Stack before call

previous contents	
-- *rectPtr* --	**Long**—POINTER to space for RECT defining rectangle to be set
left	**Word**—INTEGER; left X coordinate for rectangle
top	**Word**—INTEGER; top Y coordinate for rectangle
right	**Word**—INTEGER; right X coordinate for rectangle
bottom	**Word**—INTEGER; bottom Y coordinate for rectangle
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetRect(rectPtr,left,top,right,bottom)
Rect *rectPtr;
Integer     left;
Integer     top;
Integer     right;
Integer     bottom;
```

$6B04 SetRectRgn

Destroys previous region information by setting a specified region to a specified rectangle. If the inputs do not describe a valid rectangle, the region is set to the empty region. If the original region was not rectangular, the region is resized.

Parameters

Stack before call

previous contents	
-- *rgnHandle* --	**Long**—HANDLE to region being set
left	**Word**—INTEGER; left X coordinate for rectangle
top	**Word**—INTEGER; top Y coordinate for rectangle
right	**Word**—INTEGER; right X coordinate for rectangle
bottom	**Word**—INTEGER; bottom Y coordinate for rectangle
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Memory Manager errors Returned unchanged

C

```
extern pascal void SetRectRgn(rgnHandle,left,top,right,bottom)

RgnHandle     rgnHandle;

Integer     left;

Integer     top;

Integer     right;

Integer     bottom;
```

$4004 SetRgnSave

Sets the *rgnSave* field in the GrafPort to a specified value.

Warning

This is an internal routine that should not be used by application programs.

Parameters

Stack before call

previous contents
rgnSaveValue — **Long**—HANDLE; new value of *rgnSave* field
← **SP**

Stack after call

previous contents
← **SP**

Errors None

C

```
extern pascal void SetRgnSave(rgnSaveValue)
Handle     rgnSaveValue;
```

$1204 SetSCB

Sets the SCB (scan line control byte) to a specified value.

Parameters

Stack before call

previous contents	
scanLine	**Word**—INTEGER; scan line whose SCB is to be set
newSCB	**Word**—INTEGER; new value for SCB
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$0452 `badScanLine` Invalid scan line number; 0 to 199 are valid

C

```
extern pascal void SetSCB(scanLine,newSCB)
Word     scanLine;
Word     newSCB;
```

$3804 SetSolidBackPat

Sets the background pattern to a solid pattern using a specified color. Only an appropriate number of bits in *colorNum* are used. If the port SCB indicates 320 mode, four bits are used; if it indicates 640 mode, two bits are used.

Parameters

Stack before call

previous contents	
colorNum	**Word**—INTEGER; new color value
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetSolidBackPat(colorNum)

Word      colorNum;
```

$3704 SetSolidPenPat

Sets the pen pattern to a solid pattern using the specified color. Only an appropriate number of bits in *colorNum* are used. If the port SCB indicates 320 mode, four bits are used; if it indicates 640 mode, two bits are used.

Parameters

Stack before call

previous contents
colorNum — **Word**—INTEGER; new color value
← **SP**

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void SetSolidPenPat(colorNum)

Word     colorNum;
```

$9E04 SetSpaceExtra

Sets the *spExtra* field in the GrafPort to a specified value. The *spExtra* field is used by programs that are trying to justify text to a left and right boundary. When the *spExtra* field is nonzero, its value is added to the width of each space printed in a string.

Important

SetSpaceExtra uses FIXED values. You can use the Integer Math Tool Set routine FixRatio to convert values to FIXED values.

Parameters

Stack before call

Stack		
previous contents		
-- *spaceExtra* --		**Long**—FIXED; new value for *spExtra* field
	← **SP**	

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetSpaceExtra(spaceExtra)

Fixed     spaceExtra;
```

A justifying example

You want to display the words *a quick brown fox* and left- and right-justify them in a rectangle that measures 200 pixels across. You measure the string and find it to be 193 pixels long. The string has 3 spaces between words, so you divide 3 into the 7 pixels remaining (200 – 193 = 7). Thus, you set *spExtra* to 2 1/3 (7 + 3 = 2 1/3).

$8D04 SetStdProcs

Sets up a specified record of pointers for customizing QuickDraw II operations. At the time of publication, more details were unavailable.

Parameters

Stack before call

previous contents
-- *stdProcRecPtr* --

Long—POINTER to *stdProcs* record

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void SetStdProcs(stdProcRecPtr)

QDProcsPtr     stdProcRecPtr;
```

$4804 SetSysField

Sets the *sysField* field in the GrafPort to a specified value.

Warning

This is an internal routine that should not be used by application programs.

Parameters

Stack before call

previous contents	
-- *sysFieldValue* --	**Long**—New value for *sysField* field
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetSysField(sysFieldValue)

Longint     sysFieldValue;
```

$B204 SetSysFont

Sets a specified font as the system font. The default system font is used unless this call is made. A handle to the system font is placed in the *fontHandle* field of each GrafPort when it is opened or initialized.

Parameters

Stack before call

Stack	
previous contents	
-- *fontHandle* --	**Long**—HANDLE to font that will be system font
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetSysFont(fontHandle)

FontHndl      fontHandle;
```

$9A04 SetTextFace

Sets the text face to a specified value. Up to 16 operations on the text are possible. Each bit in *textFace* represents a different face, as shown in Figure 16-39.

Parameters

Stack before call

previous contents
textFace — **Word**—INTEGER; text face (see Figure 16-39)
← **SP**

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void SetTextFace(textFace)

TextStyle     textFace;
```

(continued)

Text face flag

The bit values for the *textFace* parameter are shown in Figure 16-39.

Important

Shadow, outline, and italic styles are available only if QuickDraw II Auxiliary has been loaded and started up. Also, fonts that have a descent value of less than 2 will not be underlined.

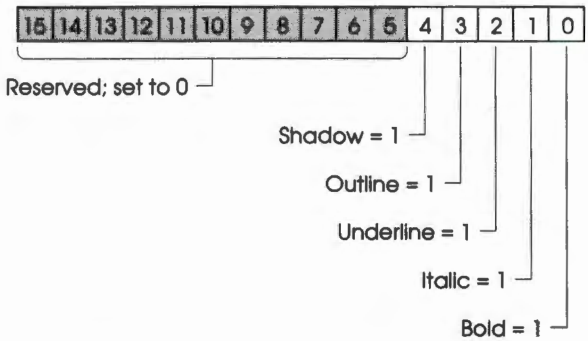


Figure 16-39
Text face flag

$9C04 SetTextMode

Sets the text mode to a specified value. There are eight text-only modes (four modes and their opposites), as shown in Table 16-10. The fastest modes are the modes that only transfer the foreground to the destination. The fastest of the foreground modes are `modeForeOR` and `modeForeXOR`; `modeForeBIC` is almost as fast, and `modeForeCOPY` is the slowest.

In addition to the text-only modes, the pen modes apply to text. See the section "SetPenMode" in this chapter.

Parameters

Stack before call

Stack	
previous contents	
textMode	**Word**—INTEGER; text mode (see Table 16-10)
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetTextMode(textMode)

Word     textMode;
```

(continued)

Text modes

The modes shown in Table 16-10 are used only for text. They apply when drawing from a one-bit-per-pixel world to a two- or four-bit-per-pixel world. You need this routine only when drawing from the font to a destination pixel map.

Table 16-10
Text-only modes

Integer	Name	Description
$0004	`modeForeCopy`	Copies only the foreground pixels into the destination; background pixels are not altered
$8004	`notForeCOPY`	Same as `modeForeCopy`, except that foreground pixels are turned to background pixels and background pixels are turned to foreground pixels before the operation is performed
$0005	`modeForeOR`	ORs only the foreground pixels into the destination; background pixels are not altered
$8005	`notForeOR`	Same as `modeForeOR`, except that foreground pixels are turned to background pixels and background pixels are turned to foreground pixels before the operation is performed
$0006	`modeForeXOR`	XORs only the foreground pixels into the destination; background pixels are not altered
$8006	`notForeXOR`	Same as `modeForeXOR`, except that foreground pixels are turned to background pixels and background pixels are turned to foreground pixels before the operation is performed
$0007	`modeForeBIC`	BICs only the foreground pixels into the destination; background pixels are not altered
$8007	`notForeBIC`	Same as `modeForeBIC`, except that foreground pixels are turned to background pixels and background pixels are turned to foreground pixels before the operation is performed

$D204 SetTextSize

Sets the *txSize* field of the GrafPort to a specified value.

Parameters

Stack before call

previous contents
textSize — **Word**—INTEGER; value for *txSize* field
← **SP**

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void SetTextSize(textSize)

Word     textSize;
```

$4604 SetUserField

Sets the *userField* field in the GrafPort to a specified value. Your application can attach data to a GrafPort by using this field as a pointer to some other data area.

Parameters

Stack before call

previous contents
-- *userFieldValue* --
← **SP**

userFieldValue: **Long**—New value for *userField* field

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void SetUserField(userFieldValue)

Longint     userFieldValue;
```

$C804 SetVisHandle

Sets the *visRgn* field in the GrafPort to a specified value.

Parameters

Stack before call

previous contents
-- *rgnHandle* --

rgnHandle: **Long**—HANDLE to visible region

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void SetVisHandle(rgnHandle)

RgnHandle     rgnHandle;
```

$B404 SetVisRgn

Copies a specified region into the visible region (but does not change the *visRgn* field of the GrafPort).

Parameters

Stack before call

previous contents
-- *rgnHandle* --

rgnHandle: **Long**—HANDLE to region

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void SetVisRgn(rgnHandle)

RgnHandle     rgnHandle;
```

$9104 ShowCursor

Shows the cursor by incrementing the cursor level (if the level is already 0, it is not incremented). A cursor level of 0 indicates the cursor is visible; a cursor level of less than 0 indicates the cursor is not visible.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

None

C

```
extern pascal void ShowCursor()
```

$2804 ShowPen

Increments the pen level. A positive pen level indicates that drawing will occur; a negative pen level indicates that drawing will not occur.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void ShowPen()
```

$3904 SolidPattern

Sets a specified pattern to a solid pattern using a specified color. Only an appropriate number of bits in *colorNum* are used. If the port SCB indicates 320 mode, four bits are used; if it indicates 640 mode, two bits are used.

Parameters

Stack before call

Stack	
previous contents	
colorNum	**Word**—INTEGER; new color value
patternPtr	**Long**—POINTER to pattern
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors None

C

```
extern pascal void SolidPattern(colorNum,patternPtr)
Word      colorNum;
Pattern      patternPtr;
```

$AD04 StringBounds

Puts the string bounds rectangle of a specified Pascal-type string into a specified buffer.

Parameters

Stack before call

previous contents	
-- *stringPtr* --	**Long**—POINTER to Pascal-type string
-- *resultPtr* --	**Long**—POINTER to space for RECT defining rectangle
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void StringBounds(stringPtr,resultPtr)

Pointer    stringPtr;

Rect *resultPtr;
```

$A904 StringWidth

Returns the sum of all the character widths, in pixels (pen displacements), of a specified Pascal-type string. This would be the pen displacement if the string were to be drawn.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *stringPtr* --	**Long**—POINTER to Pascal-type string
	← **SP**

Stack after call

previous contents	
stringWidth	**Word**—INTEGER; width of string in pixels
	← **SP**

Errors

None

C

```
extern pascal Integer StringWidth(stringPtr)

Pointer     stringPtr;
```

$8104 SubPt

Subtracts the source point from the destination point and leaves the result in the destination point. For example, a source point of (1,2) and a destination point of (10,20) result in a destination point of (9,18).

Parameters

Stack before call

Stack	
previous contents	
— *srcPtPtr* —	**Long**—POINTER to POINT
— *destPtPtr* —	**Long**—POINTER to POINT used as source and destination
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SubPt(srcPtPtr,destPtPtr)

Point *srcPtPtr;

Point *destPtPtr;
```

$AF04 TextBounds

Puts the character bounds rectangle of specified text into a specified buffer.

Parameters

Stack before call

previous contents	
textPtr	**Long**—POINTER to text
textLength	**Word**—INTEGER; length of text in bytes
resultPtr	**Long**—POINTER to space for RECT defining rectangle
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void TextBounds(textPtr,textLength,resultPtr)

Pointer     textPtr;

Word     textLength;

Rect *resultPtr;
```

$AB04 TextWidth

Returns the character width, in pixels (pen displacement), of specified text.

Parameters

Stack before call

Stack	
previous contents	
wordspace	**Word**—Space for result
textPtr	**Long**—POINTER to text
textLength	**Word**—INTEGER; length of text in bytes
	← **SP**

Stack after call

Stack	
previous contents	
textWidth	**Word**—INTEGER; width of text in pixels
	← **SP**

Errors

None

C

```
extern pascal Integer TextWidth(textPtr,textLength)

Pointer     textPtr;

Word     textLength;
```

$4E04 UnionRect

Calculates the smallest rectangle that contains both source rectangles and places the result in a destination rectangle.

Parameters

Stack before call

previous contents	
rect1Ptr	**Long**—POINTER to first source rectangle
rect2Ptr	**Long**—POINTER to second source rectangle
unionRectPtr	**Long**—POINTER to destination rectangle
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void UnionRect(rect1Ptr,rect2Ptr,unionRectPtr)

Rect *rect1Ptr;

Rect *rect2Ptr;

Rect *unionRectPtr;
```

$7204 UnionRgn

Calculates the smallest region that contains every point that is in either source region and places the result in a destination region. The destination region (which may be one of the source regions) must already exist; UnionRgn does not allocate it.

If both regions are empty, the destination is set to an empty region.

Parameters

Stack before call

Stack	
previous contents	
-- *rgn1Handle* --	**Long**—HANDLE to one source region
-- *rgn2Handle* --	**Long**—HANDLE to another source region
-- *unionRgnHandle* --	**Long**—HANDLE to destination region
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

Memory Manager errors — Returned unchanged

C

```
extern pascal void UnionRgn(rgn1Handle,rgn2Handle,unionRgnHandle)
RgnHandle      rgn1Handle;
RgnHandle      rgn2Handle;
RgnHandle      unionRgnHandle;
```

$7404 XorRgn

Calculates the difference between the union and the intersection of two regions and places the result in a destination region. The destination region (which may be one of the source regions) must already exist; this routine does not allocate it.

If the regions are not coincident, the destination is set to an empty region.

Parameters

Stack before call

previous contents	
-- *rgn1Handle* --	**Long**—HANDLE to one source region
-- *rgn2Handle* --	**Long**—HANDLE to another source region
-- *xorRgnHandle* --	**Long**—HANDLE to destination region
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

Memory Manager errors — Returned unchanged

C

```
extern pascal void XorRgn(rgn1Handle,rgn2Handle,xorRgnHandle)

RgnHandle      rgn1Handle;

RgnHandle      rgn2Handle;

RgnHandle      xorRgnHandle;
```

QuickDraw II summary

This section briefly summarizes the constants, data structures, and tool set error codes contained in QuickDraw II.

Important

These definitions are provided in the appropriate interface file.

Table 16-11
QuickDraw II constants

Name	Value	Description
Color data		
`table320`	$32	320 color table
`table640`	$32	640 color table
GrafPort sizes		
`maskSize`	$08	Mask size
`locSize`	$10	Loc size
`patsize`	$20	Pattern size
`pnStateSize`	$32	Pen state size
`portSize`	$AA	Size of GrafPort
Color masks		
`blueMask`	$000F	Mask for blue nibble
`greenMask`	$00F0	Mask for green nibble
`redMask`	$0F00	Mask for red nibble
Font flags		
`widMaxSize`	$0001	Nonproportional spacing
`zeroSize`	$0002	Numeric spacing
Master colors		
`black`	$0000	Works in 320 and 640 modes
`blue`	$000F	Works in 320 and 640 modes
`darkGreen320`	$0080	Works in 320 mode
`green320`	$00E0	Works in 320 mode
`green640`	$00F0	Works in 640 mode
`lightBlue320`	$04DF	Works in 320 mode
`purple320`	$072C	Works in 320 mode
`darkGray320`	$0777	Works in 320 mode
`periwinkleBlue320`	$078F	Works in 320 mode
`brown320`	$0841	Works in 320 mode
`lightGray320`	$0CCC	Works in 320 mode
`red320`	$0D00	Works in 320 mode

Table 16-11 (continued)
QuickDraw II constants

Name	Value	Description
Master colors		
`lilac320`	$0DAF	Works in 320 mode
`red640`	$0F00	Works in 640 mode
`orange320`	$0F70	Works in 320 mode
`flesh320`	$0FA9	Works in 320 mode
`yellow`	$0FF0	Works in 320 and 640 modes
`white`	$0FFF	Works in 320 and 640 modes
Pen modes		
`modeCopy`	$0000	Copy source to destination
`modeOR`	$0001	Overlay source and destination
`modeXOR`	$0002	XOR pen with destination
`modeBIC`	$0003	Bit Clear pen with destination
`notCopy`	$8000	Copy (not source) to destination
`notOR`	$8001	Overlay (not source) and destination
`notXOR`	$8002	XOR (not pen) with destination
`notBIC`	$8003	Bit Clear (not pen) with destination
Pen and text modes		
`modeForeCopy`	$0004	Copy foreground pixels into destination
`modeForeOR`	$0005	OR foreground pixels into destination
`modeForeXOR`	$0006	XOR foreground pixels into destination
`modeForeBIC`	$0007	BIC foreground pixels into destination
`notForeCOPY`	$8004	Turn background to foreground, then copy foreground pixels into destination
`notForeOR`	$8005	Turn background to foreground, then OR foreground pixels into destination
`notForeXOR`	$8006	Turn background to foreground, then XOR foreground pixels into destination
`notForeBIC`	$8007	Turn background to foreground, then BIC foreground pixels into destination
Mode for QDStartUp		
`mode320`	$00	320 mode
`mode640`	$80	640 mode
SCB byte masks		
`colorTable`	$0F	Color table number
`scbReserved`	$10	Reserved for future use
`scbFill`	$20	Fill mode on
`scbInterrupt`	$40	Interrupt generated when scan line refreshed
`scbColorMode`	$80	640 mode on

(continued)

Table 16-11 (continued)
QuickDraw II constants

Name	Value	Description
Text styles		
`boldMask`	$0001	Mask for bold bit
`italicMask`	$0002	Mask for italic bit
`underlineMask`	$0004	Mask for underline bit
`outlineMask`	$0008	Mask for outline bit
`shadowMask`	$0010	Mask for shadow bit

Table 16-12
QuickDraw II data structures

Name	Offset	Type	Definition
BufDimRec (buffer sizing record)			
maxWidth	$0	Word	Application-defined maximum pixel image width
textBufHeight	$2	Word	Current text buffer height in pixels
textBufferWords	$4	Word	Current width of text buffer in words
fontWidth	$6	Word	Equal to *maxFBRExtent* used in call
Font (font record)			
offseToMF	$00	Word	Offset in number of words to Macintosh font part
family	$02	Word	Font family number
style	$04	TextStyle	Style font was designed with
size	$06	Word	Point size of font
version	$08	Word	Version number of font definition
fbrExtent	$0A	Word	Maximum horizontal distance, in pixels, to far edge of any foreground or background pixel of any character of font
FontGlobalsRecord			
fgFontID	$00	Word	Family number
fgStyle	$02	TextStyle	Style font was designed with
fgSize	$04	Word	Point size of font
fgVersion	$06	Word	Version number of font definition
fgWidMax	$08	Word	Maximum character width of any character in font
fgFBRExtent	$0A	Word	Maximum horizontal distance, in pixels, to far edge of any foreground or background pixel of any character of font
FontInfoRecord			
ascent	$00	Integer	Number of pixels above base line in font rectangle
descent	$02	Integer	Number of pixels below base line in font rectangle
widMax	$04	Integer	Maximum character width of any character in font
leading	$06	Integer	Recommended number of blank pixel rows between descent of one text line and ascent of the next

Table 16-12 (continued)
QuickDraw II data structures

Name	Offset	Type	Definition
GrafPort			
portInfo	$00	LocInfo	Location information
portRect	$10	Rect	Port rectangle
clipRgn	$18	RgnHandle	Handle to clipping region
visRgn	$1C	RgnHandle	Handle to visible region
bkPat	$20	Pattern	Background pattern
pnLoc	$40	Point	Pen location
pnSize	$44	Point	Pen size
pnMode	$48	Word	Pen mode
pnPat	$4A	Pattern	Pen pattern
pnMask	$6A	Mask	Pen mask
pnVis	$72	Word	Pen visiblility
fontHandle	$74	FontHndl	Handle to font
fontID	$78	FontID	Font ID
fontFlags	$7C	Word	Font flags
txSize	$7E	Integer	Text size
txFace	$80	TextStyle	Text face
txMode	$82	Word	Text mode
spExtra	$84	Fixed	Value of space extra
chExtra	$88	Fixed	Value of char extra
fgColor	$8C	Word	Foreground color
bgColor	$8E	Word	Background color
picSave	$90	Handle	picSave
rgnSave	$94	Handle	rgnSave
polySave	$98	Handle	polySave
grafProcs	$9C	QdProcsPtr	Pointer to GrafProcs record
arcRot	$A0	Integer	arcRot
userField	$A2	Longint	userField
sysField	$A6	Longint	sysField
LocInfo			
portSCB	$00	Word	SCB byte
ptrToPixImage	$02	Pointer	Pointer to pixel image
width	$06	Word	Width
boundsRect	$08	Rect	Boundary rectangle
PaintParam (PaintPixels parameter block)			
ptrToSourceLocInfo	$00	LocInfoPtr	Pointer to source location information
ptrToDestLocInfo	$04	LocInfoPtr	Pointer to destination location information
ptrToSourceRect	$08	RectPtr	Pointer to source rectangle
ptrToDestPoint	$0C	PointPtr	Pointer to destination point
mode	$10	Word	Mode
maskHandle	$12	Handle	Handle to clipping region

(continued)

Table 16-12 (continued)
QuickDraw II data structures

Name	Offset	Type	Definition
PenState record			
psPnSize	$00	Point	Pen size
psPnMode	$04	Word	Pen mode
psPnPat	$06	Pattern	Pen pattern
psPnMask	$26	Mask	Pen mask
ROMFontRecord			
rfFamNum	$00	Word	Font family number of ROM font
rfFamStyle	$02	Word	Style of ROM font
rfSize	$04	Word	Point size of ROM font
rfFontHandle	$06	FontHndl	Handle to font
rfNamePtr	$0A	Pointer	Pointer to font name
rfFBRExtent	$0E	Word	*fbrExtent* for ROM font

Note: The actual assembly-language equates have a lowercase letter *o* in front of all of the names given in this table.

Table 16-13
QuickDraw II error codes

Code	Name	Description
$0401	`alreadyInitialized`	Attempt made to start up QuickDraw II without first shutting it down
$0402	`cannotReset`	Never used
$0403	`notInitialized`	Quickdraw II not initialized
$0410	`screenReserved`	Memory Manager reported that screen memory (bank $E1 from $2000 to $9FFF) is already owned by someone else
$0411	`badRect`	Invalid rectangle specified
$0420	`notEqualChunkiness`	Source and destination pixels not the same type
$0430	`rgnAlreadyOpen`	Region already being saved in current GrafPort
$0431	`rgnNotOpen`	No region open in current GrafPort
$0432	`rgnScanOverflow`	Region scan overflow
$0433	`rgnFull`	Region full
$0440	`polyAlreadyOpen`	Polygon already open
$0441	`polyNotOpen`	Polygon not open
$0442	`polyTooBig`	Polygon too big
$0450	`badTableNum`	Invalid table number; 0 to 15 are valid
$0451	`badColorNum`	Invalid color number; 0 to 15 are valid
$0452	`badScanLine`	Invalid scan line number; 0 to 199 are valid
$04FF	`notImplemented`	Call not implemented

Chapter 17

QuickDraw II Auxiliary

QuickDraw II Auxiliary adds features that did not appear in QuickDraw II because of implementation time or memory space. In particular, QuickDraw II adds the capability to deal with pictures and allows applications to outline, shadow, and italicize text.

The picture routines listed in this chapter have QuickDraw II tool set numbers, and thus could be considered a part of QuickDraw II. However, those routines are not available unless QuickDraw II Auxiliary has been loaded and started up, so they are included here.

A preview of the QuickDraw II Auxiliary routines

To introduce you to the capabilities of QuickDraw II, all QuickDraw II routines are grouped by function and briefly described in Table 16-1. These routines are described in detail later in this chapter, where they are separated into housekeeping routines (discussed in routine number order) and the rest of the QuickDraw II routines (discussed in alphabetical order).

Table 17-1
QuickDraw II Auxiliary routines and their functions

Routine	Description
Housekeeping routines	
QDAuxBootInit	Initializes QuickDraw II Auxiliary; called only by the Tool Locator—must not be called by an application
QDAuxStartup	Starts up QuickDraw II Auxiliary for use by an application
QDAuxShutDown	Shuts down QuickDraw II Auxiliary when an application quits
QDAuxVersion	Returns the version number of QuickDraw II Auxiliary
QDAuxReset	Resets QuickDraw II Auxiliary; called only when the system is reset—must not be called by an application
QDAuxStatus	Indicates whether QuickDraw II Auxiliary is active
Auxiliary routines	
OpenPicture	Allocates memory for the recording of drawing commands into a picture definition and returns a handle to the picture
PicComment	Inserts a specified comment into the currently open picture
ClosePicture	Completes the picture definition process begun by an OpenPicture call
DrawPicture	Takes the drawing commands recorded in the picture definition, maps them from the picture frame into a specified destination rectangle, and draws them
KillPicture	Releases all memory occupied by a specified picture
CopyPixels	Copies a pixel image from one place to another, stretching or compressing it as necessary to make the source pixels fit the destination rectangle
DrawIcon	Draws a specified icon in a specified mode at a specified location and clips to the current visible and clipping regions
WaitCursor	Changes the cursor to a predefined cursor that looks like a watch

About pictures

A **picture,** as defined by QuickDraw II, is a record of drawing commands. The OpenPicture call establishes the picture frame that is used in conjunction with the destination rectangle to map objects from one space to another when the picture is drawn.

The mapping occurs as follows: If the picture frame is (0,0,100,100) and the destination rectangle is (50,50,60,60), a line recorded from (10,10) to (90,90) would appear from (51,51) to (59,59) when the picture is drawn.

Any text drawn is also scaled by using the picture frame and the destination rectangle. Using the previous example, a text size of 60 would appear as 6 when the picture is drawn. If the horizontal and vertical coordinates are not scaled by an equal amount, the vertical change is used to select the correct size for the font. The actual scaling of the font is handled by the Font Manager.

One of the most common uses of pictures is for printing. The Print Manager uses pictures to record what you want to put on a page, then plays back the picture over and over again into a band buffer. The band buffer is then printed. Because of this technique, your application only has to record the drawing commands once; the print driver can use the resulting picture and deal with the appropriate band buffer.

Another common use of pictures is to pass data back and forth from one application to another. The picture data type is one of two standard types defined by the Scrap Manager.

Style modification support

At the time of publication, QuickDraw II provides ROM support for bold and underlined text, while QuickDraw II Auxiliary supports outlined, shadowed, and italicized text. Therefore, if your application is using outlined, shadowed, or italicized text, or if you are allowing the user to choose such style modifications, you must load and start up QuickDraw II Auxiliary.

QuickDraw II Auxiliary icon record

An **icon** is a small graphic object that is usually symbolic of an operation or of a larger entity, such as a document. The **QuickDraw II Auxiliary icon record** indicates whether the icon is in color or black and white, the size of the icon, the height and width of the icon, the icon image, and the mask controlling the appearance of the icon, as shown in Figure 17-1.

❖ *Note:* At the time of publication, this record had not been included in the APW interface file.

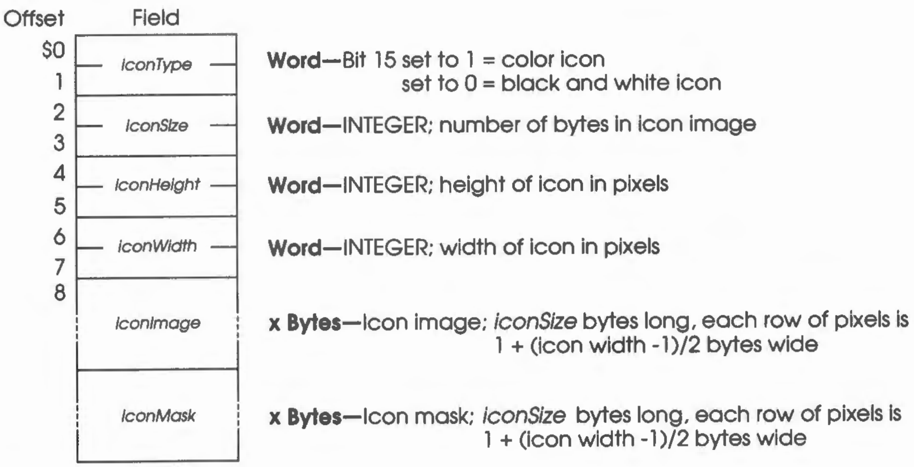


Figure 17-1
QuickDraw II Auxiliary icon record

There is also a *displayMode* word that controls how the *iconMask* is applied. When the *displayMode* word is 0, the icon is copied to the destination through the specified mask. The *displayMode* word can also have the values shown in Figure 17-2.

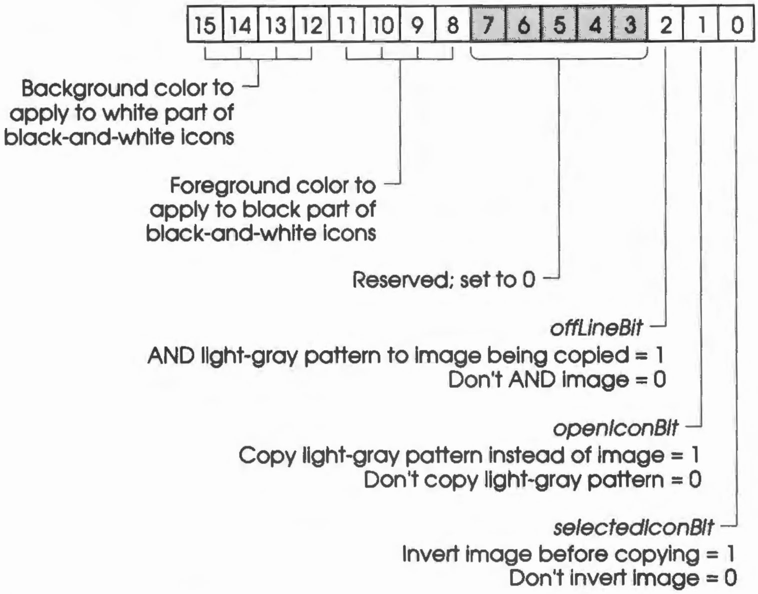


Figure 17-2
The *displayMode* word

All three of the operations in bits 2–0 can occur at once, and the testing is in the following order:

1. Check *openIconBit* (bit 1). Is it open (set to 1)?
2. Check *offLineBit* (bit 2). Is it off-line (set to 1)?
3. Check *selectedIconBit* (bit 0). Is it selected (set to 1)?

Color is only applied to the black and white icons if bits 15–8 are not all 0. Colored pixels in an icon are inverted by black pixels becoming white and any other color of pixel becoming black.

Your application draws the icon by using a DrawIcon call. See the section "DrawIcon" in this chapter.

Using QuickDraw II Auxiliary

This section discusses how the QuickDraw II Auxiliary routines fit into the general flow of an application and gives you an idea of which routines you'll need to use under normal circumstances. Each routine is described in detail later in this chapter.

QuickDraw II Auxiliary depends on the presence of the tool sets shown in Table 17-2 and requires that at least the indicated version of the tool set be present.

Table 17-2
QuickDraw II Auxiliary—other tool sets required

Tool set number		Tool set name	Minimum version needed
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0
$03	#03	Miscellaneous Tool Set	1.0
$04	#04	QuickDraw II	1.2

In addition, if your application is using the DrawPicture routine, the Font Manager (tool set number $1B) must be loaded and started up.

The first QuickDraw II Auxiliary call your application must make is QDAuxStartUp. Conversely, when you quit your application, you must make the QDAuxShutDown call.

The OpenPicture begins the picture definition process. While OpenPicture is in effect, QuickDraw II drawing commands and any comments from PicComment calls are recorded and placed in the picture definition. When your application is through recording the picture, use the ClosePicture routine to stop the picture definition process and the DrawPicture routine to draw the picture in the destination rectangle. Finally, when you are completely through with a picture, you can use the KillPicture routine to release the memory the picture occupies.

Your application can draw an icon by using a DrawIcon call, produce a cursor that looks like a watch by using a WaitCursor call, or copy a pixel image from one place to another by using a CopyPixels call.

$0112 QDAuxBootInit

Initializes QuickDraw II Auxiliary; called only by the Tool Locator.

Warning

An application must never make this call.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C Call must not be made by an application.

$0212 QDAuxStartUp

Starts up QuickDraw II Auxiliary for use by an application.

Important

Your application must make this call before it makes any other QuickDraw II Auxiliary calls.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void QDAuxStartUp()
```

$0312 QDAuxShutDown

Shuts down QuickDraw II Auxiliary when an application quits.

Important

If your application has started up QuickDraw II Auxiliary, the application must make this call before it quits.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void QDAuxShutDown()
```

$0412 QDAuxVersion

Returns the version number of QuickDraw II Auxiliary.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
versionInfo	**Word**—Version number of QuickDraw II Auxiliary
	← **SP**

Errors None

C

```
extern pascal Word QDAuxVersion()
```

$0512 QDAuxReset

Resets QuickDraw II Auxiliary; called only when the system is reset.

Warning
An application must never make this call.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C Call must not be made by an application.

$0612 QDAuxStatus

Indicates whether QuickDraw II Auxiliary is active.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
activeFlag	**Word**—BOOLEAN; TRUE if QuickDraw II Auxiliary active, FALSE if inactive
	← **SP**

Errors None

C

```
extern pascal Boolean QDAuxStatus()
```

ClosePicture

Completes the picture definition process begun by an OpenPicture call.

Important

Calls to OpenPicture and ClosePicture must be balanced; that is, one ClosePicture call must be made for every OpenPicture call.

ClosePicture calls the QuickDraw II routine ShowPen, thus balancing the HidePen call made by the OpenPicture call.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void ClosePicture()
```

$0912 CopyPixels

Copies a pixel image from one place to another, stretching or compressing it as necessary to make the source pixels fit the destination rectangle.

If the destination *locInfo* record is the same as the *locInfo* record of the current GrafPort, the pixels are also clipped to the GrafPort's visible and clipping regions.

Parameters

Stack before call

previous contents	
-- *srcLocPtr* --	**Long**—POINTER to *locInfo* record of source rectangle
-- *destLocPtr* --	**Long**—POINTER to *locInfo* record of destination rectangle
-- *srcRect* --	**Long**—POINTER to RECT defining source rectangle
-- *destRect* --	**Long**—POINTER to RECT defining destination rectangle
xferMode	**Word**—Pen mode
-- *maskRgn* --	**Long**—HANDLE to mask region
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void CopyPixels(srcLocPtr,destLocPtr,srcRect,destRect,
xferMode,maskRgn)
LocInfoPtr      srcLocPtr;
LocInfoPtr      destLocPtr;
Rect *srcRect;
Rect *destRect;
Word     xferMode;
RgnHandle     maskRgn;
```

$0B12 DrawIcon

Draws a specified icon in a specified mode at a specified location and clips to the current visible and clipping regions. The routine does not contibute to a picture definition, nor does it print the icon.

The QuickDraw II Auxiliary icon record and display mode word are described in the section "QuickDraw II Auxiliary Icon Record" in this chapter.

Parameters

Stack before call

previous contents	
— *iconPtr* —	**Long**—POINTER to icon record (see Figure 17-1)
displayMode	**Word**—Bit flag defining icon's appearance (see Figure 17-2)
xPos	**Word**—X coordinate of upper left corner of icon
yPos	**Word**—Y coordinate of upper left corner of icon
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void DrawIcon(iconPtr,displayMode,xPos,yPos)

Pointer     iconPtr;

Word      displayMode;

Word      xPos;

Word      yPos;
```

$BA04 DrawPicture

Takes the drawing commands recorded in the picture definition, maps them from the picture frame into a specified rectangle, and draws them.

Warning

If you call DrawPicture with the initial, arbitrarily large clipping region, and the destination rectangle is either offset or larger than the picture frame, the clipping region might be set to empty and no drawing will be done.

DrawPicture passes any picture comments to a low-level procedure accessed through the *grafProcs* field of the GrafPort. For more information, see the section "PicComment" in this chapter.

Parameters

Stack before call

Stack	
previous contents	
-- *picHandle* --	**Long**—HANDLE to picture
-- *destRect* --	**Long**—POINTER to RECT defining destination rectangle
← **SP**	

Stack after call

Stack	
previous contents	
← **SP**	

Errors

None

C

```
extern pascal void DrawPicture(picHandle,destRect)

Handle     picHandle;

Rect *destRect;
```

$BB04 KillPicture

Releases all memory occupied by a specified picture. Use this call only when your application is completely through with a picture.

Parameters

Stack before call

previous contents	
-- *picHandle* --	**Long**—HANDLE to picture
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void KillPicture(pichandle)

Handle     pichandle;
```

$B704 OpenPicture

Allocates memory for the recording of drawing commands into a picture definition, and returns a handle to the picture.

Warning

A GrafPort's clipping region is initialized to an arbitrarily large region. To ensure that the clipping region is still valid when drawing occurs and the region is mapped from the *picFrame* to the *destRect*, you should always change the clipping region to a smaller region before calling OpenPicture.

OpenPicture also calls the QuickDraw II routine HidePen, so no drawing will occur on the screen while the picture is open.

❖ *Note:* No drawing will occur unless you call ShowPen just after OpenPicture, or you call ShowPen without previously balancing it by a call to HidePen.

When a picture is open, the GrafPort's *picSave* field contains a handle to information related to the picture definition. If you want to temporarily disable the collection of routine calls and picture comments, save the current value of *picSave* and set the field to NIL; then restore the value when you want to resume the picture definition.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *picFrame* --	**Long**—POINTER to RECT defining picture frame
← **SP**	

Stack after call

previous contents	
-- *picHandle* --	**Long**—HANDLE to picture
← **SP**	

Errors

None

C

```
extern pascal Handle OpenPicture(picFrame)

Pointer     picFrame;
```

$B804 PicComment

Inserts a specified comment into the currently open picture. An application that processes the comments must include a procedure to do the processing and store a pointer to that procedure in the *grafProcs* field of the GrafPort.

❖ *Note:* The standard low-level procedure for processing picture comments simply ignores all comments.

Parameters

Stack before call

previous contents	
commentKind	**Word**—Type of comment
dataSize	**Word**—Size of additional data; 0 if none
dataHandle	**Long**—HANDLE to additional data; NIL if none
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void PicComment(commentKind,dataSize,dataHandle)

Integer     commentKind;

Integer     dataSize;

Handle     dataHandle;
```

$0A12 WaitCursor

Changes the cursor to a predefined cursor that looks like a watch.

❖ *Note:* You can restore the standard arrow cursor by making the QuickDraw II call InitCursor.

A desk accessory or tool set can make this without checking whether QuickDraw II Auxiliary is active and without checking for errors. If QuickDraw II Auxiliary has not been loaded and started up, the dispatcher will return an error, but nothing else will happen.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void WaitCursor()
```

QuickDraw II Auxiliary summary

QuickDraw II Auxiliary does not contain any predefined constants, data structures, or error codes.

Chapter 18

SANE Tool Set

The **Standard Apple Numeric Environment (SANE)** is extended-precision IEEE arithmetic, with elementary functions. It scrupulously conforms to the IEEE standard 754 for binary floating-point arithmetic and to the proposed IEEE standard 854, which is a radix-independent and word-length-independent standard for floating-point arithmetic.

SANE provides sufficient numeric support for most applications. It includes

- IEEE types single (32-bit), double (64-bit), and extended (80-bit)
- A 64-bit type for large integer computations, such as those used in accounting
- Fundamental floating-point operations (+ – * / √ rem)
- Comparisons
- Binary-decimal and float-integer conversions
- Scanning and formatting for ASCII numeric strings
- Logs, trigs, and exponentials
- Compound and annuity functions for financial computations
- A random-number generator
- Functions for management of the floating-point environment
- Other functions required or recommended by the IEEE standard

The SANE Tool Set fully supports the Standard Apple Numeric Environment, matching the functions of the Macintosh SANE packages, as well as those of the 6502 assembly language SANE software from which it is derived.

The SANE Tool Set comprises the usual tool set housekeeping routines and the routines SANEFP816, SANEElems816, and SANEDecStr816 that serve as entry points for the major pieces of SANE code. Each call to SANEFP816, SANEElems816, or SANEDecStr816 passes an opword parameter specifying the operation to be performed. For example, the opword $0206 passed to SANEFP816 indicates "divide by a value of type single."

This chapter describes only the basic functions of the SANE routines and the differences between the 65C816 version and the 6502 version. SANE functions are completely documented in the *Apple Numerics Manual,* which you will need if you use SANE routines in your application.

A preview of the SANE Tool Set routines

To introduce you to the capabilities of the SANE Tool Set, all SANE Tool Set routines are grouped by function and briefly described in Table 18-1. These routines are described in detail later in this chapter, where they are separated into housekeeping routines (discussed in routine number order) and the rest of the SANE Tool Set routines (discussed in alphabetical order).

Table 18-1
SANE Tool Set routines and their functions

Routine	Description
Housekeeping routines	
SANEBootInit	Initializes the SANE Tool Set; called only by the Tool Locator—must not be called by an application
SANEStartup	Starts up the SANE Tool Set for use by an application
SANEShutDown	Shuts down the SANE Tool Set when an application quits
SANEVersion	Returns the version number of the SANE Tool Set
SANEReset	Resets the SANE Tool Set; called only when the system is reset—must not be called by an application
SANEStatus	Indicates whether the SANE tool set is active
SANE routines	
SANEDecStr816	Contains numeric scanners and formatter
SANEElems816	Contains elementary functions, financial functions, and a random-number generator
SANEFP816	Contains basic arithmetic operations, comparsions, conversions, environmental control, and IEEE auxiliary operations

Using the SANE Tool Set

This section discusses how the SANE Tool Set routines fit into the general flow of an application and gives you an idea of which routines you'll need to use under normal circumstances. Each routine is described in detail later in this chapter.

The SANE Tool Set depends upon the presence of the tool sets shown in Table 18-2 and requires that at least the indicated version of the tool set be present.

Table 18-2
SANE Tool Set—other tool sets required

Tool set number		Tool set name	Minimum version needed
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0

The first SANE Tool Set call that your application must make is SANEStartUp. Conversely, when you quit your application, you must call SANEShutDown.

You can program with the SANE tool set using any assembler that generates code for the Apple IIGS. The equate file SANE.equs and the macro file M16.SANE help you use SANE with the Apple IIGS Programmer's Workshop (APW) assembler.

❖ *Note:* The APW C Compiler fully supports SANE and includes C interfaces to all SANE tool set routines.

The following code for a binary operation illustrates a typical invocation of a SANE function:

```
PUSHLONG      <4-byte source operand address>

PUSHLONG      <4-byte destination operand address>

PUSHWORD      <OpWord>

(SANE macro)
```

Some SANE operations require different numbers of arguments, some pass 16-bit integer arguments by value, and some return results in the X, Y, and status registers.

The following example illustrates the use of the numeric scanner and formatter. The procedure accepts as an argument an ASCII string representing a number of degrees and returns the trigonometric sine of the argument as a numeric ASCII string. Both input and output are Pascal strings; that is, byte 0 gives the length, and byte 1 contains the first character in the string. The caller of the procedure pushes the address of the input string and performs a JSR to SINE. The procedure overwrites the input string with the result (whose length may be as large as 80) and clears the stack. SINE uses SANEFP816, SANEElems816, and SANEDecStr816, which are the three principal functions in the SANE Tool Set.

```
; Somewhere early in the program, initialize SANE
; Call the Memory Manager to reserve 256 bytes of bank $0 for use
;    as SANE direct page. #SANEdirectpg is the address of this memory in this example
         PUSHWORD   #SANEdirectpg
         _SANEStartUp
; Near the end of the program, shut down SANE.
         _SANEShutDown
; Call the Memory Manager to release the memory reserved for the
;    SANE direct page (often by releasing ALL reserved memory)
; Procedure SINE (var s : DecStr)
;    s :        I/O string
;    index :    16-bit integer index
;    theDec :   Decimal record
;    vp :       Boolean for validprefix
;    theForm :  decform record
;    x :        Extended temporary
;    const :    Extended constant = pi/180
;
SINE     ENTRY
         PLA
         STA        return                 ; Save return address
         PLA
         STA        sAdr                   ; Address of s -> sAdr
         PLA
         STA        sAdr+2
         LDA        #1
         STA        index                  ; 1 -> index
         PUSHLONG   sAdr
         PUSHLONG   #index
         PUSHLONG   #theDec
         PUSHLONG   #vp
         FPSTR2DEC                         ; s -> theDec
```

```
         PUSHLONG   #theDec
         PUSHLONG   #x
         FDEC2X                         ; theDec -> x
         PUSHLONG   #const
         PUSHLONG   #x
         FMULX                          ; Convert to radians:  x*const -> x
         PUSHLONG   #x
         FSINX                          ; Sin(x) -> x
         PUSHLONG   #theForm
         PUSHLONG   #x
         PUSHLONG   #theDec
         FX2DEC                         ; x -> theDec
         PUSHLONG   #theForm
         PUSHLONG   #theDec
         PUSHLONG   sAdr
         FDEC2STR                       ; theDec -> s
         LDA        return
         PHA
         RTS
index    ds         2
vp       ds         2
const    dc         h'AE C8 E9 94 12 35 FA 8E F9 3F'  ; Constant = pi/180
x        ds         10
theForm  dc         i'1,10'             ; First style, then digits
theDec   ds         33                  ; Sign, exponent, length, ASCII = (2+2+1+28)
return   ds         2
sAdr     ds         4
```

Performance characteristics and limitations

Your application must preserve bytes 24 to 29 (decimal) of the SANE direct page between calls to SANE. Those 6 bytes hold the floating-point environment and halt vector. The remainder of the SANE direct page is scratch space used only during SANE execution and does not need to be preserved across calls to SANE. The space is thus available to the application.

Warning
Future implementations of SANE on the IIGS may not store the floating-point environment and halt vector on the direct page. If you access these variables directly, you may forfeit upward compatibility. Always access these variables only through SANE calls.

Except for the SANEVersion routine, SANE Tool Set routines remove all arguments from the stack and return no results on the stack. Temporary stack growth during calls does not exceed 40 bytes (50 for elementary function calls).

The SANE Tool Set conforms to the general tool set rules for management of the CPU registers, modes, and busy flag. However, SANE never returns tool set error codes; all floating-point errors are handled internally by setting exception flags (see the *Apple Numerics Manual*).

Typical timings, based on a few sample values, are as follows:

0.5 – 1.2 ms	Add, subtract
1.0 – 5.0 ms	Multiply
1.9 – 5.2 ms	Divide
0.7 – 1.4 ms	Scanner
0.8 – 1.0 ms	Formatter
2.4 – 5.8 ms	Extended-to-decimal
0.8 – 3.5 ms	Decimal-to-extended
50 – 100 ms	Trigonometric, exponential, logarithmic

Differences between 65C816 and 6502 SANE

The 65C816 version of SANE differs from the 6502 version in several ways. First, all address parameters in the 65C816 are 4 bytes instead of 2 bytes. Another difference is that, if you are not using macro calls, you access the routines via the tool set dispatcher rather than via a JSR statement. Thus, invocations end with

```
LDX     #SANEtsNum + FuncNum*256
JSL     $E10000
```

instead of

```
JSR     <xx6502>
```

If you are using the macro calls, you don't see this difference, because the macro automatically expands into the correct version of the call.

The low-order bytes of the X and Y registers return information as documented in Part II of the *Apple Numerics Manual.* The high-order byte of X duplicates the contents of the low-order byte of Y. The high-order byte of Y is undefined. Figure 18-1 shows the relationship between the return information in 6502 registers and that in 65816 registers.

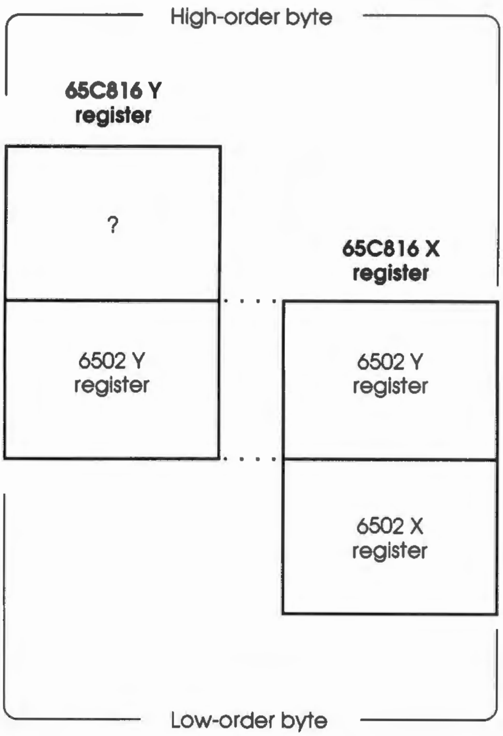


Figure 18-1
SANE return information

The Remainder call uses the N flag from the Processor Status register and both bits 7 and 15 of the X register to return the sign of the quotient. The low 7 bits of X contain the absolute value of the quotient.

The final difference between the 65816 and 6502 versions lies in the halt mechanism. When a halt occurs in 65816 SANE, the input parameters and SANE opcode are located in the SANE direct page, as shown in Figure 18-2. For one- and two-argument calls, C holds the address of DST. For two-argument calls, B holds the address of SRC. For binary-to-decimal conversion, A holds the address of the decimal record, B holds the address of the binary value, and C holds the address of the decform record.

Important

Halts occur only on calls to SANEFP816. SANEElems816 stimulates the halt mechanism only through the procexit call. SANEDecStr816 makes no calls to SANEFP816 and therefore never stimulates the halt mechanism.

The **halt vector** is the address of a halt-handling routine stored as a 4-byte address. The SetHaltVector routine expects the 4-byte halt vector to be passed by value on the stack. GetHaltVector returns a 3-byte halt vector in the X and Y registers. X contains the two low bytes (first and second) of the halt vector, and Y contains the second and third bytes; the second byte of the halt vector occurs in both the X and Y registers.

Figure 18-2 illustrates how X and Y return the halt vector.

Offset	Field
$0, 1, 2, 3, 4, 5	2 return addresses
6, 7	Caller's direct page
8, 9	Caller's data bank
0A, 0B	Opword
0C, 0D, 0E, 0F	"C" address
10, 11, 12, 13	"B" address
14, 15, 16, 17	"A" address
18, 19, 1A, 1B	Halt vector
1C, 1D	Environment
1E, 1F	Pending exceptions
20	Pending X-lo
21	Pending X-hi, Y-lo
22	Pending Y-hi

Figure 18-2
SANE direct page on halt

When a halt occurs, the 65816 SANE executes a JSL to HaltVector. The halt handler can continue execution as if no halt had occurred by executing RTL.

Important

SANE is not reentrant! Your application can't call SANE within the halt handler. During the halt, the direct page can't be altered, and making a SANE call would alter the direct page.

After return to the SANEFP816 code, SANE uses the A register, not *pending exceptions*, to set the final floating-point exceptions. Execute LDA PendingExceptions at the end of the halt handler to ensure that the exceptions from the current call are handled correctly.

Warning

Pending exceptions is a sum of the five exception constants, represented as an integer from 0 to 31. Unpredictable results occur if the A register contains a value outside this range when your application exits the halt handler.

$010A SANEBootInit

Initializes the SANE Tool Set; called only by the Tool Locator.

Warning
An application must never make this call.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C Call must not be made by an application.

$020A SANEStartUp

Starts up the SANE Tool Set for use by an application. This routine clears the SANE environment word, installing the default settings of round-to-nearest, round-to-extended-precision, all exceptions clear, and all halts disabled (these terms are defined in the *Apple Numerics Manual).*

Important

If you are using assembly language, your application must make this call before making any other SANE calls. If you are using APW C, the call is made automatically.

SANEStartUp also sets the halt vector to 0. As with 6502 SANE, halts remain inoperative until the application both enables halts and also gives the halt vector a nonzero address.

The *dPageAddr* parameter is the address of one page (256 bytes) in bank $0 that your application makes available to the SANE Tool Set for its use.

Parameters

Stack before call

previous contents	
dPageAddr	**Word**—Bank $0 starting address of one page of direct-page space
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

The call is made automatically in APW C.

$030A SANEShutDown

Shuts down the SANE Tool Set.

Important

If your application has started up the SANE Tool Set, the application must make this call before it quits.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void SANEShutDown()
```

$040A SANEVersion

Returns the version number of the SANE Tool Set.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
versionInfo	**Word**—Version number of SANE Tool Set
	← **SP**

Errors None

C

```
extern pascal Word SANEVersion()
```

$050A SANEReset

Resets the SANE Tool Set; called only when the system is reset.

Warning

An application must never make this call.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C Call must not be made by an application.

$060A SANEStatus

Indicates whether the SANE Tool Set is active.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
activeFlag	**Word**—BOOLEAN; TRUE if SANE Tool Set active, FALSE if inactive
	← **SP**

Errors None

C

```
extern pascal Boolean SANEStatus()
```

$0A0A SANEDecStr816

Contains numeric scanners and formatter.

Parameters See the *Apple Numerics Manual* for details.

Errors None

C

```
extern pascal Void SANEDecStr816()
```

$0B0A SANEElems816

Contains elementary functions, financial functions, and a random-number generator.

Parameters See the *Apple Numerics Manual* for details.

Errors None

C

```
extern pascal Void SANEElems816()
```

$090A SANEFP816

Contains basic arithmetic operations, comparisons, conversions, environmental control, and IEEE auxiliary operations.

Parameters See the *Apple Numerics Manual* for details.

Errors None

C

```
extern pascal Void SANEFP816()
```

SANE Tool Set summary

The constants and data structures for the SANE Tool Set are defined in the *Apple Numerics Manual*. The SANE Tool Set does not contain any tool set error codes.

Chapter 19

Scheduler

The **Scheduler** delays activation of a desk accessory or of other tasks until the resources that the desk accessory or task needs become available. Much of the system code is not **reentrant;** that is, the code cannot be called while it is already executing. For example, if a desk accessory was activated while the system was within nonreentrant code, the system would most likely fail. To prevent such a situation from occurring, the Apple IIGS provides a busy flag that the Scheduler can check to see if a needed resource is busy or available.

If you are writing an application or a desk accessory, you won't need to use the Scheduler. The only time you need to use the Scheduler is when you are writing one of the following:

- Your own tool set
- Interrupt handlers that access ProDOS 16 or the tool set routines

For example, an application that performed background printing might need to access the Scheduler.

A preview of the Scheduler routines

To introduce you to the capabilities of the Scheduler, all Scheduler routines are grouped by function and briefly described in Table 19-1. These routines are described in detail later in this chapter, where they are separated into housekeeping routines (discussed in routine number order) and the rest of the Scheduler routines (discussed in alphabetical order).

Table 19-1
Scheduler routines and their functions

Routine	Description
Housekeeping routines	
SchBootInit	Initializes the Scheduler; called only by the Tool Locator—must not be called by an application
SchStartUp	Starts up the Scheduler for use by an application
SchShutDown	Shuts down the Scheduler when an application quits
SchVersion	Returns the version number of the Scheduler
SchReset	Resets the Scheduler; called only when the system is reset—must not be called by an application
SchStatus	Indicates whether the Scheduler is active
Scheduler queue routines	
SchAddTask	Adds a task to the Scheduler queue
SchFlush	Flushes all tasks in the Scheduler queue—must not be called by an application

Using the Scheduler

This section discusses how the Scheduler routines fit into the general flow of an application and gives you an idea of which routines you'll need to use under normal circumstances. Each routine is described in detail later in this chapter.

The Scheduler depends upon the presence of the tool sets shown in Table 19-2 and requires that at least the indicated version of the tool set be present.

Table 19-2
Scheduler—other tool sets required

Tool set number	Tool set name	Minimum version needed
$01 #01	Tool Locator	1.0

Your application should make a SchStartUp call before making any other Scheduler calls.

❖ *Note:* At the time of publication, the SchStartUp call was not an absolute requirement, because the Tool Locator automatically started up the Scheduler at boot time. However, you should make the call anyway to guarantee that your application remains compatible with all future versions of the system.

Your application should also call SchShutDown when the application quits.

The Scheduler revolves around the **busy flag** located at $E1/00FF. If you wish to change the state of the busy flag, you should use the routines INCBUSYFLG and DECBUSYFLG. Those routines are not tool set routines, but are accessed directly from vectors in bank $E1 as follows:

INCBUSYFLG	$E1/0064
DECBUSYFLG	$E1/0068

When a nonreentrant module is entered, your tool set or interrupt handler should perform a JSL to INCBUSYFLG in full native mode (e, m, and x flags all set to 0). When exiting from the module, the application should perform a JSL to DECBUSYFLG. DECBUSYFLG decrements the busy flag and executes any tasks placed in the Scheduler queue with the tool set routine SchAddTask.

Your tool set or interrupt handler should use SchAddTask after it checks the state of the busy flag. If the flag is set to other than 0, the necessary system resources are not currently available, and you can add a task to the Scheduler queue by using the SchAddTask routine.

Note that your application should never call the SchFlush routine.

$0107 SchBootInit

Initializes the Scheduler; called only by the Tool Locator.

Warning
An application must never make this call.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C Call must not be made by an application.

$0207 SchStartUp

Starts up the Scheduler for use by an application.

❖ *Note:* At the time of publication, the SchStartUp call was not an absolute requirement, because the Tool Locator automatically started up the Scheduler at boot time. However, you should make the call anyway, to guarantee that your application remains compatible with all future versions of the system.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void SchStartUp()
```

$0307 SchShutDown

Shuts down the Scheduler when an application quits.

Important

If your application has started up the Scheduler, the application must make this call before it quits.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

None

C

```
extern pascal void SchShutDown()
```

$0407 SchVersion

Returns the version number of the Scheduler.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
versionInfo	**Word**—Version number of Scheduler
	← **SP**

Errors

None

C

```
extern pascal Word SchVersion()
```

$0507 SchReset

Resets the Scheduler; called only when the system is reset.

Warning

An application must never make this call.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C Call must not be made by an application.

$0607 SchStatus

Indicates whether the Scheduler is active.

Parameters

Stack before call

previous contents
wordspace — **Word**—Space for result
← **SP**

Stack after call

previous contents
activeFlag — **Word**—BOOLEAN; TRUE if Scheduler active, FALSE if inactive
← **SP**

Errors None

C

```
extern pascal Boolean SchStatus()
```

$0907 SchAddTask

Adds a task to the Scheduler queue. The queue has space for four items, enough to support the Desk Manager as well as other small interrupt handlers.

The Scheduler uses a JSL to launch the procedure pointed to by *taskPtr*. If the task can't be added to the queue because the queue is already full, *onQueueFlag* will be FALSE.

❖ *Note:* The Scheduler is not designed to support multitasking.

When the busy flag is decremented to 0, the tasks in the queue are executed in the posted order.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
— *taskPtr* —	**Long**—POINTER to task to be added
	← **SP**

Stack after call

previous contents	
onQueueFlag	**Word**—BOOLEAN; TRUE if task added to Scheduler's queue, FALSE if queue was full
	← **SP**

Errors

None

C

```
extern pascal Boolean SchAddTask(taskPtr)

VoidProcPtr     taskPtr;
```

$0A07 SchFlush

Flushes all tasks in the Scheduler's queue.

Important
An application must never make this call.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C Call must not be made by an application.

Scheduler summary

The Scheduler does not contain any predefined constants, data structures, or tool set error codes.

Chapter 20

Scrap Manager

The **Scrap Manager** lets an application handle cutting and pasting. From the user's point of view, all data that is cut or copied resides in the Clipboard. The Cut command deletes data from a document and places it in the Clipboard. The Copy command copies data into the Clipboard without deleting it from the document. A subsequent Paste command will insert the contents of the Clipboard at a specified place, whether that place is in the same document or another or in the same application or another. An application that supports cutting and pasting may also provide a Clipboard window for displaying the current contents of the scrap; it may show the Clipboard window at all times or only when requested via the Show (or Hide) Clipboard command.

❖ *Note:* The Scrap Manager is intended to transfer limited amounts of data; attempts to transfer very large amounts of data may fail due to lack of memory.

The nature of the data to be transferred varies according to the application. For example, in a word processor, the data is text; in a graphics application, it's a picture. The amount of information retained about the data being transferred also varies. Between two text applications, text can be cut and pasted without any loss of information; however, if the user of a graphics application cuts a picture consisting of text and then pastes it into a word processor document, the text in the picture may not be editable in the word processor, or it may be editable but not look exactly the same as in the graphics application. The Scrap Manager allows for a variety of data types and provides a mechanism whereby applications have some control over how much information is retained when data is transferred.

The desk scrap is usually stored in memory, but can be stored on the disk (in the file CLIPBOARD in the SYSTEM subdirectory of the boot volume) if there's not enough room for it in memory.

❖ *Macintosh programmers:* The scrap does not have to be in memory when an application starts or stops.

A preview of the Scrap Manager routines

To introduce you to the capabilities of the Scrap Manager, all Scrap Manager routines are grouped by function and briefly described in Table 20-1. These routines are described in detail later in this chapter, where they are separated into housekeeping routines (discussed in routine number order) and the rest of the Scrap Manager routines (discussed in alphabetical order).

Table 20-1
Scrap Manager routines and their functions

Routine	Description
Housekeeping routines	
ScrapBootInit	Initializes the Scrap Manager; called only by the Tool Locator—must not be called by an application
ScrapStartUp	Starts up the Scrap Manager for use by an application
ScrapShutDown	Shuts down the Scrap Manager when an application quits
ScrapVersion	Returns the version number of the Scrap Manager
ScrapReset	Resets the Scrap Manager; called only when the system is reset—must not be called by an application
ScrapStatus	Indicates whether the Scrap Manager is active
Scrap routines	
UnloadScrap	Writes the desk scrap from memory to the scrap file and releases the memory it occupied
LoadScrap	Reads the desk scrap from the scrap file into memory
ZeroScrap	Clears the contents of the scrap and increments the scrap count
PutScrap	Appends specified data to the scrap that has the same scrap type as the data
GetScrap	Copies scrap information of the appropriate type to a specified handle, setting the handle to the correct size
GetScrapCount	Returns the current scrap count
GetScrapState	Returns a flag indicating the current state of the scrap
GetScrapHandle	Returns a copy of the handle for the scrap of a specified type
GetScrapSize	Returns the size of the specified scrap
GetScrapPath	Returns a pointer to the pathname used for the Clipboard file
SetScrapPath	Sets a pointer to the pathname used for the Clipboard file

Memory and the desk scrap

A large desk scrap can prevent an application from being loaded. If your application needs to know about whether there's enough room for the desk scrap in memory, you can set up your application so that a small initial segment of it is loaded. That segment can contain a Scrap Manager call to get the scrap size.

After a decision is made about whether to keep the scrap in memory or on disk, the remaining segments of the application can be loaded as needed. Of course, if there isn't enough room for the scrap at application load time, there probably won't be room for it later when a user tries to paste its contents into a document.

There are other disadvantages to keeping the desk scrap on disk: The disk may be locked, it may not have enough room for the scrap, or it may be removed during use of the application.

Important
If the application can't write the scrap to disk, it should put up an alert box informing the user, who may want to cancel the operation at that point.

Desk scrap data types

From the user's point of view there can be only one item on the Clipboard at a time, but the application may store more than one version of the information in the scrap, each representing the same Clipboard contents in a different form. For example, text cut from a word processor document may be stored in the desk scrap both as text and as a QuickDraw II picture.

Why would you want to do this? You might want your application to keep information in its own internal format, but you may also want it to be able to communicate via the Clipboard with other applications. When a user cuts or copies something to the Clipboard, the application can put it there in two different ways:

1. It can put it there internally so that a subsequent paste operation can be easily handled.
2. It can put it there publicly so that if the user tries to paste it into another application or desk accessory, the other application can handle it.

There are two public scrap types, as shown in Table 20-2.

Table 20-2
Public scrap types

Value	Name	Scrap type
0	textScrap	Text
1	picScrap	Picture

Applications must write at least one of these standard types of data to the desk scrap and must be able to read both types. Most applications will prefer one of these types over the other; for example, a word processor prefers text, whereas a graphics application prefers pictures. An application should write at least its preferred standard type of data to the desk scrap, and it may write both types (to pass the most information possible to the receiving application, which may prefer the other type).

An application reading the desk scrap looks for its preferred data type. If the application's preferred type isn't there—or if it's there but it was written by an application having a different preferred type—the receiving application may or may not be able to convert the data to the type it needs. If it cannot, some information may be lost in the transfer process. For example, a graphics application can easily convert text to a picture, but the reverse isn't true.

Using the Scrap Manager

This section discusses how the Scrap Manager routines fit into the general flow of an application and gives you an idea of which routines you'll need under normal circumstances. Each routine is described in detail later in this chapter.

The Scrap Manager depends upon the presence of the tool sets shown in Table 20-3 and requires that at least the indicated version of the tool set be present.

Table 20-3
Scrap Manager — other tool sets required

Tool set number		Tool set name	Minimum version needed
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0

The first Scrap Manager call that your application must make is ScrapStartUp. Conversely, when you quit your application, you must call ScrapShutDown.

If your application supports display of the Clipboard, you can call GetScrapCount to find out whether a desk accessory has changed the desk scrap. The **scrap count** indicates how many times the scrap has changed. Save the value of this field when one of your application's windows is deactivated and a system window is activated. Check the value each time through the main event loop to see whether it has changed; if it has, the contents of the desk scrap have changed. If the Clipboard window is visible, it needs to be updated whenever the count changes.

When the user gives a Cut or Copy command, your application needs to write the cut or copied data to the desk scrap. First call ZeroScrap to clear its previous contents; then call PutScrap to put the data into the scrap. If it makes it easier for your application to transfer data, you can call PutScrap more than once with the same scrap type.

When the user gives a Paste command, call GetScrap to access data of a particular type in the desk scrap and to get information about the data.

❖ *Note:* ZeroScrap, PutScrap, and GetScrap all keep track of whether the scrap is in memory or on the disk, so you don't have to worry about loading it first. After any of these calls, the scrap will be in memory again.

Setting up a private scrap

Instead of using the desk scrap for storing data that's cut and pasted within an application, you may want to set up a **private scrap** for this purpose.

❖ *Note:* In most applications that use the standard text or picture data types, it's simpler for the application to use the desk scrap. However, if your application defines its own private type of data, or if very large amounts of data might be cut and pasted, using a private scrap may result in faster cutting and pasting within the application.

The format of a private scrap can be anything you want, because no other application will use it. For example, an application can simply maintain a pointer to cut or copied data. The application must, however, be able to convert data between the format of its private scrap and the format of the desk scrap.

❖ *Note:* The LineEdit scrap is a private scrap for applications that use LineEdit. LineEdit provides routines that access its own scrap and transfer data between the LineEdit scrap and the desk scrap.

If you use a private scrap, you must be sure that the right data is always pasted when the user gives a Paste command (the right data being whatever was most recently cut or copied in any application or desk accessory) and that the Clipboard, if visible, always shows the current data. You should copy the contents of the desk scrap to your private scrap at application startup and whenever a desk accessory is deactivated (call GetScrap to access the desk scrap). When the application is terminated, or when a desk accessory is activated, you should copy the contents of the private scrap to the desk scrap. Call ZeroScrap to clear its previous contents; call PutScrap to write data to the desk scrap.

If transferring data between the two scraps means converting it, and possibly losing information, you can copy the scrap only when you actually need to, at the time something is cut or pasted. The desk scrap needn't be copied to the private scrap unless one of the following conditions is true:

- A Paste command is given before the first Cut or Copy command after the application starts up.
- A desk accessory that changed the scrap was deactivated.

Until then, you must keep the contents of the desk scrap intact. If the Clipboard window is visible, you must display the desk scrap, instead of the private scrap, in that window.

After one of the preceding conditions has occurred, you can ignore the desk scrap until a desk accessory is activated or the application is terminated; in either of these cases, you must copy the private scrap back to the desk scrap. Thus, whatever was last cut or copied within the application is pasted if a Paste command is given in a desk accessory or in the next application. If no Cut or Copy commands are given within the application, you never have to change the desk scrap.

If your application encounters problems in trying to copy one scrap to another, it should alert the user. If the desk scrap is too large to copy to the private scrap, the user may want to leave the application or simply proceed with an empty Clipboard. If the private scrap is too large to copy to the desk scrap, either because it's disk based and too large to copy into memory or because it exceeds the maximum size allowed for the desk scrap, the user may want to stay in the application and cut or copy a smaller item.

$0116 ScrapBootInit

Initializes the Scrap Manager; called only by the Tool Locator.

Warning
An application must never make this call.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C Call must not be made by an application.

$0216 ScrapStartUp

Starts up the Scrap Manager for use by an application.

Important
Your application must make this call before it makes any other Scrap Manager calls.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void ScrapStartUp()
```

$0316 ScrapShutDown

Shuts down the Scrap Manager.

Important

If your application has started up the Scrap Manager, the application must make this call before it quits.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void ScrapShutDown()
```

$0416 ScrapVersion

Returns the version number of the Scrap Manager.

Parameters

Stack before call

Stack	
previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

Stack	
previous contents	
versionInfo	**Word**—Version number of the Scrap Manager
	← **SP**

Errors None

C

```
extern pascal Word ScrapVersion()
```

$0516 ScrapReset

Resets the Scrap Manager; called only when the system is reset.

Warning
An application must never make this call.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C Call must not be made by an application.

$0616 ScrapStatus

Indicates whether the Scrap Manager is active.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
activeFlag	**Word**—BOOLEAN; TRUE if Scrap Manager active, FALSE if inactive
	← **SP**

Errors None

C

```
extern pascal Boolean ScrapStatus()
```

$0D16 GetScrap

Copies scrap information of the appropriate type to a specified handle, setting the handle to the correct size.

❖ *Note:* To copy the desk scrap to the LineEdit scrap, use the LineEdit routine LEFromScrap.

Parameters

Stack before call

previous contents	
— *destHandle* —	**Long**—HANDLE to scrap destination
scrapType	**Word**—Scrap type; `picScrap`, `textScrap`, or application defined
← **SP**	

Stack after call

Stack
previous contents
← **SP**

Errors

$1610	badScrapType	No scrap of this type found
Memory Manager errors		Returned unchanged
ProDOS errors		Returned unchanged

C

```
extern pascal void GetScrap(destHandle,scrapType)

Handle     destHandle;

Word     scrapType;
```

$1216 GetScrapCount

Returns the current scrap count. The count changes every time ZeroScrap is called. You can use this count for testing whether the contents of the desk scrap have changed; if ZeroScrap has been called, presumably PutScrap has also been called. This information may be useful if your application supports display of the Clipboard or has a private scrap.

Parameters

Stack before call

Stack	
previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

Stack	
previous contents	
scrapCount	**Word**—INTEGER; current scrap count
	← **SP**

Errors

None

C

```
extern pascal unsigned int GetScrapCount()
```

$0E16 GetScrapHandle

Returns a copy of the handle for the scrap of a specified type.

GetScrapHandle allows you to access the scrap without making a copy of it, which might be important when memory is in short supply.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
scrapType	**Word**—Scrap type; `picScrap`, `textScrap`, or application defined
← **SP**	

Stack after call

previous contents	
-- *scrapHandle* --	**Long**—HANDLE to specified type of scrap
← **SP**	

Errors

$1610	`badScrapType`	No scrap of this type found
Memory Manager errors		Returned unchanged
ProDOS errors		Returned unchanged

C

```
extern pascal Handle GetScrapHandle(scrapType)
Word     scrapType;
```

$1016 GetScrapPath

Returns a pointer to the pathname used for the Clipboard file.

Parameters

Stack before call

previous contents		
-- *longspace* --	**Long**—Space for result	
	← **SP**	

Stack after call

previous contents		
-- *pathPtr* --	**Long**—POINTER to pathname	
	← **SP**	

Errors

None

C

```
extern pascal Pointer GetScrapPath()
```

$0F16 GetScrapSize

Returns the size of a specified scrap.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
scrapType	**Word**—Scrap type; `picScrap`, `textScrap`, or application defined
← **SP**	

Stack after call

previous contents	
-- *scrapSize* --	**Long**—Size of scrap in bytes
← **SP**	

Errors

$1610	`badScrapType`	No scrap of this type found
Memory Manager errors		Returned unchanged
ProDOS errors		Returned unchanged

C

```
extern pascal LongWord GetScrapSize(scrapType)
Word     scrapType;
```

$1316 GetScrapState

Returns a flag indicating the current state of the scrap. The *scrapState* flag is set to a nonzero value if the scrap is in memory; it is set to 0 if the scrap is currently on disk.

❖ *Note:* The *scrapState* flag is actually 0 if the scrap *should* be on disk. The scrap may not be on disk because a user can delete the Clipboard file.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
scrapState	**Word**—INTEGER; nonzero if scrap in memory, 0 if on disk
	← **SP**

Errors

None

C

```
extern pascal Word GetScrapState()
```

$0A16 LoadScrap

Reads the desk scrap from the scrap file into memory. If the desk scrap is already in memory, it does nothing. If the Clipboard file cannot be found, no error is returned; the computer responds as if you had loaded an empty Clipboard file.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

Memory Manager errors	Returned unchanged
ProDOS errors	Returned unchanged

C

```
extern pascal void LoadScrap()
```

$0C16 PutScrap

Appends specified data to the scrap that has the same type as the data. If the scrap is on disk, the scrap is loaded.

Important
Don't forget to call ZeroScrap if you want to clear the scrap's previous contents. If you don't call ZeroScrap, the data is appended to the existing scrap.

❖ *Note:* To copy the LineEdit scrap to the desk scrap, use the LineEdit routine `LEToScrap`.

Parameters

Stack before call

previous contents	
numBytes	**Long**—LONGINT; number of bytes to write
scrapType	**Word**—Scrap type; `picScrap`, `textScrap`, or application defined
srcPtr	**Long**—POINTER to data to be placed in scrap
← **SP**	

Stack after call

Stack
previous contents
← **SP**

Errors

Memory Manager errors	Returned unchanged
ProDOS errors	Returned unchanged

C

```
extern pascal void PutScrap(numBytes,scrapType,srcPtr)

unsigned Longint     numBytes;

Word      scrapType;

Pointer      srcPtr;
```

$1116 SetScrapPath

Sets a pointer to the pathname used for the Clipboard file.

Parameters

Stack before call

previous contents
-- *pathPtr* --
← **SP**

pathPtr: **Long**—POINTER to pathname of Clipboard

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void SetScrapPath(pathPtr)

Pointer     pathPtr;
```

$0916 UnloadScrap

Writes the desk scrap from memory to the scrap file and releases the memory the desk scrap occupied. If the desk scrap is already on disk, UnloadScrap does nothing.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

Memory Manager errors	Returned unchanged
ProDOS errors	Returned unchanged

C

```
extern pascal void UnloadScrap()
```

$0B16 ZeroScrap

Clears the contents of the scrap, whether the scrap is memory or on disk, and also changes the scrap count. When the user selects Cut or Copy, your application should call ZeroScrap before it calls PutScrap.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

Memory Manager errors	Returned unchanged
ProDOS errors	Returned unchanged

C

```
extern pascal void ZeroScrap()
```

Scrap Manager summary

This section briefly summarizes the constants and tool set error codes contained in the Scrap Manager. There are no predefined data structures for the Scrap Manager.

Important

These definitions are provided in the appropriate interface file.

Table 20-4
Scrap Manager constants

Name	Value	Description
Public scrap type		
`textScrap`	$0000	Text scrap
`picScrap`	$0001	Picture scrap

Table 20-5
Scrap Manager error codes

Code	Name	Description
$1610	`badScrapType`	No scrap of this type found

Chapter 21

Sound Tool Set

The **Sound Tool Set** gives you the ability to access the sound hardware without having to know specific hardware input/output addresses.

Sound Tool Set calls (other than the standard housekeeping routines) can be broken down into two groups. The first group is made through the normal tool call mechanism, with parameters being passed to and from the called routines on the stack. The second group is composed of low-level routines that, unlike most tool calls, use an eight-bit accumulator, pass their parameters in registers, and are accessed through a jump table.

There are two other tool sets dealing with sound—the Note Synthesizer and the Note Sequencer. These tool sets are not documented in the *Apple IIGS Toolbox Reference*. At the time of publication, documentation of these tool sets is planned for another, as yet unnamed, book.

❖ *Note:* This chapter uses the terms *Note Synthesizer* and *Free-Form Synthesizer*. In the world of Apple sound, these terms refer to specific tool sets and not to electronic musical instruments.

A preview of the Sound Tool Set routines

To introduce you to the capabilities of the Sound Tool Set, all Sound Tool Set routines are grouped by function and briefly described in Table 21-1. These routines are described in detail later in this chapter, where they are separated into housekeeping routines (discussed in routine number order), the rest of the numbered routines (discussed in alphabetical order), and the low-level routines.

Important

The low-level routines do not have tool set routine names and thus do not fit the alphabetical order organization; instead, they are organized in jump table offset order. Also note that the low-level routines do not have routine numbers.

Table 21-1
Sound Tool Set routines and their functions

Routine	Description
Housekeeping routines	
SoundBootInit	Initializes the Sound Tool Set; called only by the Tool Locator—must not be called by an application
SoundStartUp	Starts up the Sound Tool Set for use by an application
SoundShutDown	Shuts down the Sound Tool Set
SoundVersion	Returns the version number of the Sound Tool Set
SoundReset	Resets the Sound Tool Set; called only when the system is reset—must not be made by an application
SoundToolStatus	Indicates whether the Sound Tool Set is active
RAM and volume routines	
WriteRamBlock	Writes a specified number of bytes from system RAM into DOC RAM
ReadRamBlock	Reads a specified number of bytes from from DOC RAM into system RAM
GetSoundVolume	Reads the volume setting for a generator
SetSoundVolume	Changes the volume setting for the volume registers in the DOC or changes the system volume
GetTableAddress	Returns the jump table address for the low-level routines
Free-Form Synthesizer routines	
FFStartSound	Enables the DOC to start generating sound on a particular generator
FFStopSound	Halts any specified sound generators that are generating sound
FFSoundStatus	Returns the status of all 15 sound generators
FFGeneratorStatus	Reads the first two bytes of the generator control block corresponding to a specified generator
SetSoundMIRQV	Sets up the entry point into the sound-interrupt handler
SetUserSoundIRQV	Sets up the entry point for an application-defined synthesizer interrupt handler
FFSoundDoneStatus	Returns the current Free-Form Synthesizer playing status
Low-level routines	
Read Register	Reads any register in the DOC
Write Register	Writes a onebyte parameter to any register in the DOC
Read RAM	Reads any specified DOC RAM location
Write RAM	Writes a one-byte value to any specified DOC RAM location
Read Next	Reads the next location pointed to by the Sound GLU address register
Write Next	Writes one byte of data to the next DOC register or RAM location, depending on the setting of the Sound GLU control register
Disable Increment	Disables the auto-increment mode set up by a Read Register, Write Register, Read RAM, or Write RAM low-level sound routine, thus allowing your application to read a DOC register or memory location continuously

Sound hardware

The Apple IIGS sound hardware supports two sound subsystems. The first subsystem is an extension of the Apple IIe sound capabilities. Using this subsystem, applications toggle a soft switch, which in turn generates clicks in a speaker. In addition, the IIGS allows the application to control the volume of the speaker.

The second subsystem uses a digital oscillator chip (DOC) and 64K of dedicated RAM. The Sound Tool Set contains all of the firmware routines required to access the sound hardware. Figure 21-1 shows the major functional blocks of the sound hardware.

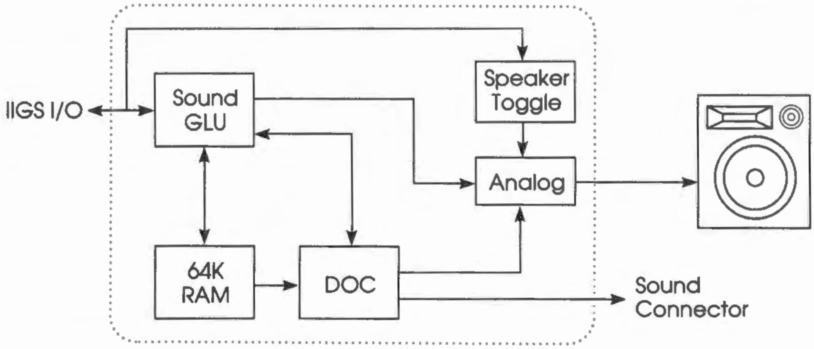


Figure 21-1
Sound hardware block diagram

The **sound GLU** (general logic unit) acts as the interface chip between system hardware and sound hardware. Figure 21-2 shows the sound GLU registers.

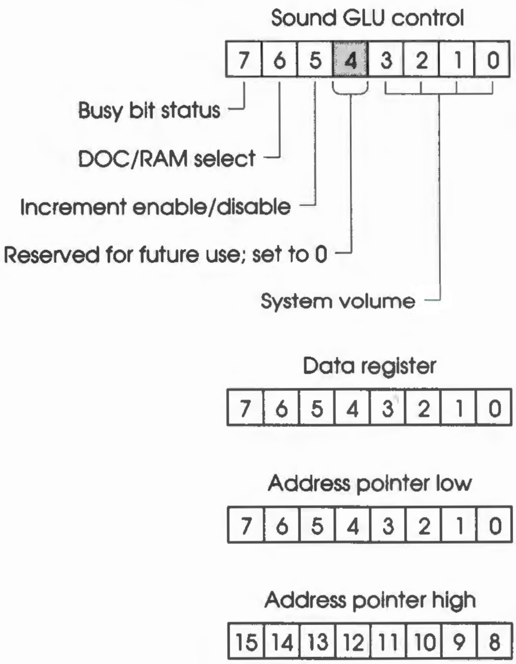


Figure 21-2
Sound GLU registers

The DOC RAM stores the waveforms used for sound generation. The DOC can create sounds of any pitch and duration. Table 21-2 shows the DOC registers.

Table 21-2
DOC register allocation

Register		Bits							
number	**Function**	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
$00–1F	Frequency low	FL7	FL6	FL5	FL4	FL3	FL2	FL1	FL0
$20–3F	Frequency high	FH7	FH6	FH5	FH4	FH3	FH2	FH1	FH0
$40–5F	Volume	V7	V6	V5	V4	V3	V2	V1	V0
$60–7F	Data sample	W7	W6	W5	W4	W3	W2	W1	W0
$80–9F	Waveform table pointer	P7	P6	P5	P4	P3	P2	P1	P0
$A0–BF	Control	CA3	CA2	CA1	CA0	1E	M2	M1	H
$C0–DF	Bank select/table size/resolution	X	BS	T2	T1	T0	R2	R1	R0
$E0	Oscillator interrupt	IRQ	1	04	03	02	01	00	1
$E1	Oscillator enable	X	X	E4	E3	E2	E1	E0	X
$E2	Analog/digital converter	S7	S6	S5	S4	S3	S2	S1	S0

For further information on the DOC, see the *Apple IIGS Hardware Reference.*

The analog section contains all the circuitry needed to amplify and filter the signal coming from the sound GLU or the DOC. The signal will be sent to the speaker.

The sound connector provides the connection to interface cards that can take the tones generated by the DOC and modify them further. Two examples of possible sound cards are programmable filter stereo interface cards and sound sampling cards.

Oscillators and generators

An oscillator is the basic sound-generating unit in the DOC. The DOC contains 32 oscillators, each of which can function independently from all the other oscillators.

One of the modes of the DOC is called swap mode. The Free-Form Synthesizer uses this mode to generate sounds. In swap mode, a pair (or **swap pair**) of oscillators form a functional oscillator unit called a **generator.** There are 15 generators defined in the Apple IIGS sound system. An oscillator-to-generator translation table converts an oscillator number to the appropriate generator number. That table is accessed through the jump table shown in the section "GetTableAddress" in this chapter.

Each oscillator controls seven DOC registers in the range 00–DF, as shown in Table 21-3.

Table 21-3
Oscillator registers

Register function	Oscillator 0 registers	Oscillator 1 registers	Oscillator *n* registers	. . .	Oscillator 31 registers
Frequency low	$00	$01	$00 + *n*	. . .	$1F
Frequency high	$20	$21	$20 + *n*	. . .	$3F
Volume control	$40	$41	$40 + *n*	. . .	$5F
Data sampling	$60	$61	$60 + *n*	. . .	$7F
Waveform table pointer	$80	$81	$80 + *n*	. . .	$9F
Control register	$A0	$A1	$A0 + *n*	. . .	$BF
Bank select/table size/resolution	$C0	$C1	$C0 + *n*	. . .	$DF

Oscillators 30 and 31 are reserved for system use and should not be used by applications. If an interrupt is generated by oscillator 30 or 31, control is passed to the System Failure Manager with an "unclaimed sound interrupt" message. When the SoundBootInit routine is called by the system at boot time, all of the sound interrupt handler pointers point to the System Failure Manager with this message.

The work area for the sound routines (specified in the SoundStartUp call) is broken down into 16 groups of 16 bytes each, with each 16-byte group comprising one **generator control block (GCB)**. The first byte of each contains the synthesizer type being used by that generator. The high nibble is reserved for use by the system and must be zero. The low nibble of the byte (bits 3–0) contains the type. The remaining 15 bytes in the GCB are for use by the application, and their meaning and value may vary depending on the synthesizer type.

Using the Sound Tool Set

This section discusses how the Sound Tool Set routines fit into the general flow of an application and gives you an idea of which routines you'll need to use under normal circumstances. Each routine is described in detail later in this chapter.

The Sound Tool Set depends on the presence of the tool sets shown in Table 21-4, and requires that at least the indicated version of the tool set be present.

Table 21-4
Sound Tool Set—other tool sets required

Tool Set number		Tool Set name	Minimum version needed
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0

Your application must make the SoundStartUp call before it makes any other Sound Tool Set calls. Conversely, when your application quits, it must make the SoundShutDown call.

The Sound Tool Set gives you the ability to control the sound hardware without having to access the hardware registers directly. To provide this capability, the Sound Tool Set must be able to read and write to RAM, read and write to the DOC registers, and raise and lower the volume. This requires a set of low-level sound routines. Unlike the other Sound Tool Set routines, which use the stack to pass parameters in the normal tool call fashion, these routines use registers to pass parameters.

❖ *Note:* Because the low-level sound routines have been designed and implemented to increase performance, they may not be available in all programming languages. At the time of publication, for example, the low-level sound routines were not available from Apple IIGS Workshop C.

The low-level routines are entered through a jump table. The table address can be obtained through a call to the GetTableAddress routine. The actual format of the jump table is shown in the section "GetTableAddress" in this chapter.

$0108 SoundBootInit

Initializes the Sound Tool Set; called only by the Tool Locator.

Warning
An application must never make this call.

This routine performs the following:

- Resets all of the DOC RAM
- Resets the Sound Tool Set's work area
- Resets the oscillators to an uninitialized state

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C Call must not be made by an application.

$0208 SoundStartUp

Starts up the Sound Tool Set for use by an application. The direct page must be page-aligned and locked until the SoundShutDown call is made.

Important
Your application must make this call before it makes any other Sound Tool Set calls.

Parameters

Stack before call

previous contents
dPageAddr
← SP

dPageAddr: **Word**—Bank $0 starting address of one page of direct-page space

Stack after call

previous contents
← SP

Errors

$0810	`noDOCFndErr`	No DOC or RAM found
$0818	`sndAlreadyStrtErr`	Sound tools already started

C

```
extern pascal void SoundStartUp(dPageAddr)

Word     dPageAddr;
```

$0308 SoundShutDown

Shuts down the Sound Tool Set.

Important

If your application has started up the Sound Tool Set, the application must make this call before it quits.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void SoundShutDown()
```

$0408 SoundVersion

Returns the version number of the Sound Tool Set.

Parameters

Stack before call

Stack	
previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

Stack	
previous contents	
versionInfo	**Word**—Version number of Sound Tool Set
	← **SP**

Errors None

C

```
extern pascal Word SoundVersion()
```

$0508 SoundReset

Resets the Sound Tool Set; called only when the system is reset.

Warning

An application must never make this call. This call is used only by the system to control the shutdown of generators. If you want to shut down a generator, use the StopSound routine.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C Call must not be made by an application.

$0608 SoundToolStatus

Indicates whether the Sound Tool Set is active.

Parameters

Stack before call

Stack	
previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

Stack	
previous contents	
activeFlag	**Word**—BOOLEAN; TRUE if Sound Tool Set active, FALSE if inactive
	← **SP**

Errors None

C

```
extern pascal Boolean SoundToolStatus()
```

$1108 FFGeneratorStatus

Reads the first two bytes of the generator control block that corresponds to a specified generator.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
genNumber	**Word**—Number of generator whose status will be returned
	← **SP**

Stack after call

previous contents	
genStatus	**Word**—Status of *genNumber* (see Figure 21-3)
	← **SP**

Errors

None

C

```
extern pascal Word FFGeneratorStatus(genNumber)

Word     genNumber;
```

(continued)

Generator status word

The status returned in the *genStatus* parameter is in the format shown in Figure 21-3.

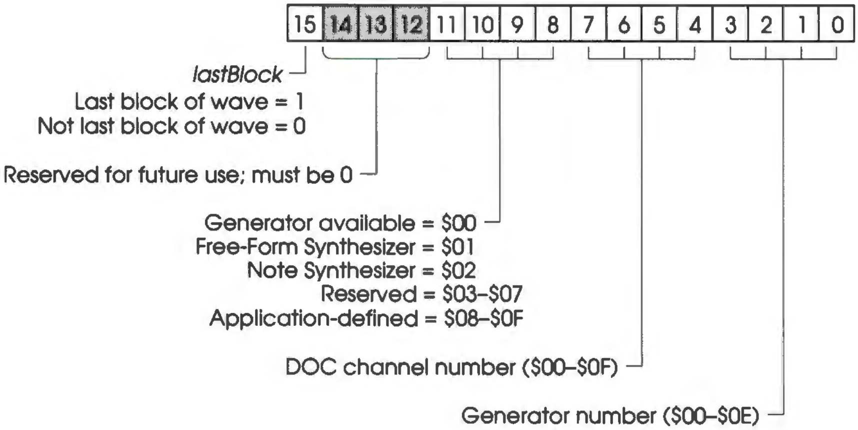


Figure 21-3
Generator status word

$1408 FFSoundDoneStatus

Returns the current Free-Form Synthesizer playing status. If the specified generator is currently playing out a waveform, the status returned to the caller will be TRUE. If the generator is not playing, the status will be FALSE ($FFFF).

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
genNumber	**Word**—Number of generator whose status will be returned
	← **SP**

Stack after call

previous contents	
genDoneFlag	**Word**—BOOLEAN; status of *genNumber*: TRUE if done playing, FALSE if playing
	← **SP**

Errors

$0813 `invalGenNumErr` Invalid generator number

C

```
extern pascal Boolean FFSoundDoneStatus(genNumber)
Word     genNumber;
```

$1008 FFSoundStatus

Returns the status of all 15 sound generators. Any bit position set to 1 in the status word returned from the function call signifies that the corresponding generator is active. The format of the status word returned is the same as that of the stop-sound mask as illustrated in Figure 21-6 in the section "FFStopSound," with bit 0 corresponding to generator 0, bit 1 corresponding to generator 1, and so on.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
genStatusInfo	**Word**—Status of all generators (bit 0 = generator 0 and so on)
	← **SP**

Errors

None

C

```
extern pascal Word FFSoundStatus()
```

$0E08 FFStartSound

Enables the DOC to start generating sound on a particular generator. If this call is made to a generator that is already active, the previous sound-generation process is terminated and the new sound process is started.

Parameters

Stack before call

Stack	
previous contents	
genNumFFSynth	**Word**—Channel, generator, and type (see Figure 21-4)
-- *pBlockPtr* --	**Long**—POINTER to parameter block for sound (see Figure 21-5)
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

$0812	`noSAppInitErr`	No SoundStartUp call made
$0813	`invalGenNumErr`	Invalid generator number
$0814	`synthModeErr`	Synthesizer mode error
$0815	`genBusyErr`	Generator already in use

C

```
extern pascal void FFStartSound(genNumFFSynth,pBlockPtr)

Word     genNumFFSynth;

Pointer     pBlockPtr;
```

(continued)

Channel-generator-type word

The values for the channel-generator-type word are shown in Figure 21-4.

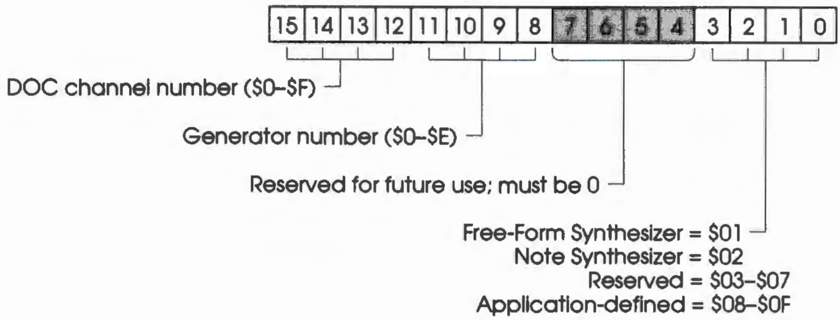


Figure 21-4
Channel-generator-type word

Parameter block

The values for the parameter block are shown in Figure 21-5. The effective output sample rate can be calculated as follows:

freqOffset = ((32 x Output sample rate in hertz)/1645)

For more detailed information on these settings, refer to the *Apple IIGS Hardware Reference.*

Offset	Field	
$0–3	*waveStart*	**Long**—Starting address of wave
4–5	*waveSize*	**Word**—Waveform size in pages
6–7	*freqOffset*	**Word**—Output sample rate
8–9	*docBuffer*	**Word**—DOC buffer start address; high-order byte significant, low-order byte = 0
0A–0B	*bufferSize*	**Word**—DOC buffer size; high-order byte = 0, low-order byte significant
0C–0F	*nextWavePtr*	**Long**—POINTER to start of next wave's parameter block
10–11	*volSetting*	**Word**—DOC volume setting; high-order byte = 0, low-order byte significant

Figure 21-5
Sound parameter block

$0F08 FFStopSound

Halts any specified sound generators that are generating sound. Depending on the setting of a 16-bit mask passed as a parameter to the routine, any of 15 generators will be stopped if running. Each bit position in the stop-sound mask corresponds to a sound generator. Bit 0 corresponds to generator 0, bit 1 corresponds to generator 1, and so on, up to bit 15 (bit 15 must be 0). Figure 21-6 illustrates the stop-sound mask.

Parameters

Stack before call

previous contents	
genMask	**Word**—Mask to stop generators (see Figure 21-6)
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void FFStopSound(genMask)

Word     genMask;
```

Stop-sound mask

The values for the stop-sound mask are shown in Figure 21-6.

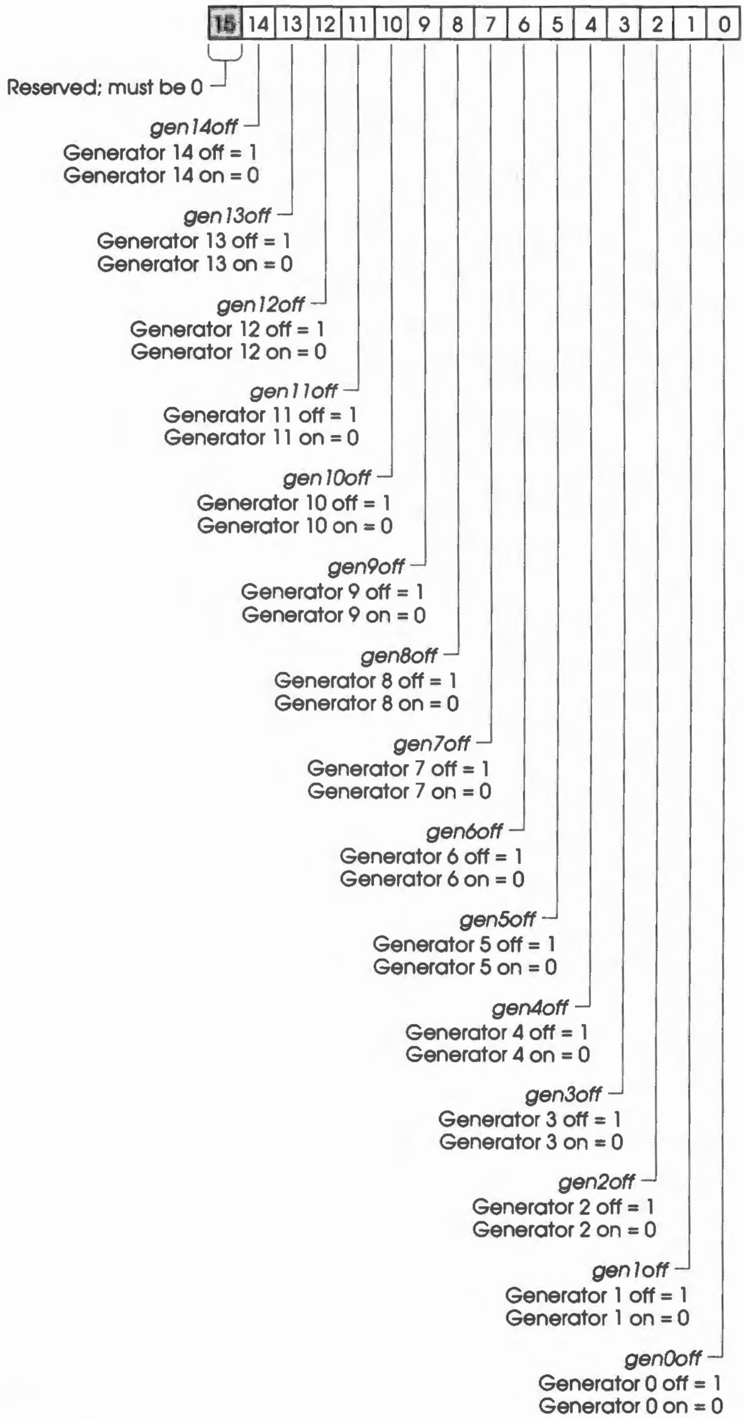


Figure 21-6
Stop-sound mask

Assembly-language example

The following example stops generators 0 and 8:

```
PEA     $1001
_StopSound
```

$0C08 GetSoundVolume

Reads the volume setting for a generator. The range of possible values is from $00 to $FF. All eight bits are valid for DOC volume registers.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
genNumber	**Word**—Number of generator whose volume will be returned
	← **SP**

Stack after call

previous contents	
volSetting	**Word**—Volume setting, from $00 to $FF; high nibble of low-order byte significant
	← **SP**

Errors

None

C

```
extern pascal Word GetSoundVolume(genNumber)
Word     genNumber;
```

$0B08 GetTableAddress

Returns the jump table address for the low-level routines (see Table 21-5).

Besides the offsets to the low-level routines, the jump table contains the following three additional functions:

- The oscillator table translates from generator number to oscillator number and returns the even number of the pair of oscillators.
- The GCB address table points to the first location of the GCB corresponding to a generator.
- The generator table translates from oscillator number to generator number.

Parameters

Stack before call

Stack	
previous contents	
-- *longspace* --	**Long**—Space for result
	← **SP**

Stack after call

Stack	
previous contents	
-- *jumpTableAddr* --	**Long**—Jump table address for low-level routines
	← **SP**

Errors

None

C

```
extern pascal Pointer GetTableAddress()
```

Jump table addresses

Table 21-5 illustrates the format of the jump table addresses used by the Sound Tool Set low-level routines.

Table 21-5
Jump table addresses for Sound Tool Set low-level outines

Routine	Offset	Constant	Address format			
			8 bits	8 bits	8 bits	8 bits
Read Register	$00	`readRegister`	Address low	Address high	Bank	$00
Write Register	$04	`writeRegister`	Address low	Address high	Bank	$00
Read RAM	$08	`readRam`	Address low	Address high	Bank	$00
Write RAM	$0C	`writeRam`	Address low	Address high	Bank	$00
Read Next	$10	`readNext`	Address low	Address high	Bank	$00
Write Next	$14	`writeNext`	Address low	Address high	Bank	$00
Oscillator table	$18	`oscTable`	Address low	Address high	Bank	$00
Generator table	$1C	`genTable`	Address low	Address high	Bank	$00
GCB address table	$20	`gcbAddrTable`	Address low	Address high	Bank	$00
Disable increment	$24	`disableInc`	Address low	Address high	Bank	$00

$0A08 ReadRamBlock

Reads a specified number of bytes from DOC RAM area into system RAM.

Warning

Interrupts must be disabled whenever your application accesses the DOC RAM. Your application must disable interrupts before it accesses the RAM and then reenable them afterward.

If the number of bytes and the starting location add up to a value greater than 64K, an error is generated.

Important

Your application must call the Memory Manager to allocate the buffer for data read from the DOC RAM.

Parameters

Stack before call

previous contents	
destPtr	**Long**—POINTER to starting RAM address where data will be written
docStart	**Word**—Starting DOC address from which data will be read
byteCount	**Word**—Number of bytes to be copied
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

$0810	`noDOCFndErr`	No DOC or RAM found
$0811	`docAddrRngErr`	DOC address range error

C

```
extern pascal void ReadRamBlock(destPtr,docStart,byteCount)

Pointer     destPtr;

Word      docStart;

Word      byteCount;
```

$1208 SetSoundMIRQV

Sets up the entry point into the sound interrupt handler. This routine is accessed every time an interrupt is generated by the DOC.

Parameters

Stack before call

previous contents
sMasterIRQ

Long—New master sound IRQ vector

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void SetSoundMIRQV(sMasterIRQ)

LongWord      sMasterIRQ;
```

$0D08 SetSoundVolume

Changes the volume setting for the volume registers in the DOC or changes the system volume.

If *genNumber* is specified as $00–$0E, the call sets the volume on the corresponding pair of generators in the DOC. If *genNumber* is specified as $0F or greater, the call sets the system volume control. The range of values for the volume setting are $00–$FF. The generator volume registers use all eight bits of resolution. The system volume control uses only the upper nibble to determine the setting.

Parameters

Stack before call

previous contents	
volume	**Word**—Generator or system volume setting; $00–$FF
genNumber	**Word**—Generator whose volume will be set; $00–$0E sets DOC, $0F–$FF sets system volume
← **SP**	

Stack after call

Stack
previous contents
← **SP**

Errors

None

C

```
extern pascal void SetSoundVolume(volume,genNumber)

Word     volume;

Word     genNumber;
```

$1308 SetUserSoundIRQV

Sets up the entry point for an application-defined synthesizer interrupt handler. When an interrupt occurs for an application-defined synthesizer, control is passed to the RAM-based synthesizer code through this vector. The old vector installed is passed back to the caller who must preserve the vector.

If control is passed to the user vector, the application-defined interrupt handler must validate that the synthesizer mode matches the synthesizer mode used by this handler. If it does not match, then the handler must pass control farther down the chain through the vector that was preserved. Control is passed through a JSL instruction; therefore, the application must return control through an RTL instruction.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *userIRQVector* --	**Long**—New user sound IRQ vector
	← **SP**

Stack after call

previous contents	
-- *oldIRQVector* --	**Long**—Old user sound IRQ vector
	← **SP**

Errors

None

C

```
extern pascal Pointer SetUserSoundIRQV(userIRQVector)

LongWord     userIRQVector;
```

$0908 WriteRamBlock

Writes a specified number of bytes from system RAM into DOC RAM.

Warning

Interrupts must be disabled whenever your application accesses the DOC RAM. Your application must disable interrupts before it accesses the RAM and then reenable them afterward.

If the sum of the starting address and the byte count is greater than 64K, an error will be returned.

Warning

Do not include the I/O space in banks $00, $01, $E0, and $E1 in the source-address range of bytes to be written to the DOC RAM. If you do, you will access soft switches that will cause the system to crash.

Parameters

Stack before call

Stack	
previous contents	
-- *srcPtr* --	**Long**—POINTER to data to be written from RAM
docStart	**Word**—Starting address of DOC buffer to receive data
byteCount	**Word**—Number of bytes to be written
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

$0810	`noDOCFndErr`	No DOC or RAM found
$0811	`docAddrRngErr`	DOC address range error

C

```
extern pascal void WriteRamBlock(srcPtr,docStart,byteCount)

Pointer     srcPtr;

Word     docStart;

Word     byteCount;
```

(None) Read register

Reads any register within the DOC. This call is made through the appropriate jump table address provided by a GetTableAddress call. (See the section "GetTableAddress" in this chapter.) The table provided by GetTableAddress includes a table of generator-to-oscillator translations. This table gives the number of the first oscillator in the pair for a generator. The number of the second oscillator equals the number of the first oscillator plus 1.

By getting the oscillator number that corresponds to a particular generator and adding it to the base register number of a register group, an application can find out the settings for an oscillator. A table of the DOC registers has been provided in the section "Sound Hardware" in this chapter.

This routine leaves the sound GLU in auto-increment and register access modes.

Parameters

The stack is not affected by this call. Instead, the following registers are used:

Relevant registers before call

e = 0; native mode
m = 1; 8-bit accumulator
x = 0; 16-bit index registers
X = DOC register to read

Relevant registers after call

Accumulator (8-bit) = Contents of specified DOC register

Errors

None

C

Call cannot be made from C.

(None) Write register

Writes a one-byte parameter to any register in the DOC. The call is made through the appropriate jump table address provided by a GetTableAddress call. To write to an oscillator register that corresponds to a generator, take the following steps:

1. Take the oscillator number from the oscillator table.
2. Add 1 to access the odd oscillator of the pair.
3. Add the base register of the specific register.
4. Make the Write Register call through the Write Register routine's address in the jump table.

This routine leaves the sound GLU in auto-increment and register access modes.

Parameters

The stack is not affected by this call. Instead, the following registers are used:

Relevant registers before call

e = 0; native mode
m = 1; 8-bit accumulator
x = 0; 16-bit index registers
Accumulator (8-bit) = data to write
X = DOC register number

Relevant registers after call

None

Errors

None

C

Call cannot be made from C.

(None) Read RAM

Reads any specified DOC RAM location.

Warning

Interrupts must be disabled whenever your application accesses the DOC RAM. Your application must disable interrupts before it accesses the RAM and then reenable them afterward.

This call is made through the appropriate jump table address provided by a GetTableAddress call. See the section "GetTableAddress" in this chapter. This routine leaves the sound GLU in auto-increment and RAM access modes.

Important

This routine does not do any type of error checking on the address or data.

Parameters The stack is not affected by this call. Instead, the following registers are used:

Relevant registers before call

e = 0; native mode
m = 1; 8-bit accumulator
x = 0; 16-bit index registers
X = DOC RAM address to read

Relevant registers after call

None

Errors None

C Call cannot be made from C.

(None) Write RAM

Writes a one-byte value to any specified DOC RAM location.

Warning

Interrupts must be disabled whenever your application accesses the DOC RAM. Your application must disable interrupts before it accesses the RAM and then reenable them afterward.

This call is made through the appropriate jump table address provided by a GetTableAddress call. See the section "GetTableAddress" in this chapter. This routine leaves the sound GLU in auto-increment and RAM access modes.

Important

This routine does not do any type of error checking on the address or data.

Parameters

The stack is not affected by this call. Instead, the following registers are used:

Relevant registers before call

e = 0; native mode
m = 1; 8-bit accumulator
x = 0; 16-bit index registers
Accumulator (8-bit) = data to write
X = DOC RAM address to write to

Relevant registers after call

None

Errors

None

C

Call cannot be made from C.

(None) Read Next

Reads the next location pointed to by the sound GLU address register.

Warning

Interrupts must be disabled whenever your application accesses the DOC RAM. Your application must disable interrupts before it accesses the RAM and then reenable them afterward.

This call is made through the appropriate jump table address provided by a GetTableAddress call.

Important

Before making the first Read Next call in a sequence, you must make a Read Register or Read RAM call with the register or address desired minus 2. This leaves the sound GLU control register set to auto-increment mode and the sound GLU address register pointing to the correct DOC register or address.

Parameters The stack is not affected by this call. Instead, the registers listed below are used:

Relevant registers before call

None (all will have been set properly by the previous Read Register or Read RAM call)

Relevant registers after call

Accumulator (8-bit) = Data byte read

Errors None

C Call cannot be made from C.

(None) Write Next

Writes one byte of data to the next DOC register or RAM location, depending on the setting of the sound GLU control register.

Warning

Interrupts must be disabled whenever your application accesses the DOC RAM. Your application must disable interrupts before it accesses the RAM and then reenable them afterward.

Important

Before making the first Write Next call in a sequence, you must make a Read Register or Read RAM call with the register or address desired minus 2. This leaves the sound GLU control register set to auto-increment mode and the sound GLU address register pointing to the correct DOC register or address.

Parameters

The stack is not affected by this call. Instead, the following registers are used:

Relevant registers before call

Accumulator (8-bit) = Data byte to write
All others will have been set properly by the previous Read Register or Read RAM call

Relevant registers after call

None

Errors

None

C

Call cannot be made from C.

(None) Disable Increment

Disables the auto-increment mode set up by a Read Register, Write Register, Read RAM, or Write RAM low-level sound routine, thus allowing your application to read a DOC register or memory location continuously. Auto-increment mode remains disabled until your application makes another Read Register, Write Register, Read RAM, or Write RAM call.

For example, if you want to read the analog-to-digital converter, your application can make a Read Register call to Register $E2 and then make a Disable Increment call. Because auto-increment mode is disabled, your application can then make a Read Next call to read the A-to-D converter continuously.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C Call cannot be made from C.

Sound Tool Set summary

This section briefly summarizes the constants, data structures, and tool set errors contained in the Sound Tool Set.

Important

These definitions are provided in the appropriate interface file.

Table 21-6
Sound Tool Set constants

Name	Value	Description
Jump table offsets		
`readRegister`	$00	Read Register routine
`writeRegister`	$04	Write Register routine
`readRam`	$08	Read RAM routine
`writeRam`	$0C	Write RAM routine
`readNext`	$10	Read Next routine
`writeNext`	$14	Write Next routine
`oscTable`	$18	Pointer to oscillator table
`genTable`	$1C	Pointer to generator table
`gcbAddrTable`	$20	Pointer to GCB address table
`disableInc`	$24	Disable Increment routine
Channel-generator-type word		
`ffSynthMode`	$0001	Free-Form Synthesizer mode
`noteSynthMode`	$0002	Note Synthesizer mode
Stop-sound mask		
`gen0off`	$0001	Generator 0 off
`gen1off`	$0002	Generator 1 off
`gen2off`	$0004	Generator 2 off
`gen3off`	$0008	Generator 3 off
`gen4off`	$0010	Generator 4 off
`gen5off`	$0020	Generator 5 off
`gen6off`	$0040	Generator 6 off
`gen7off`	$0080	Generator 7 off
`gen8off`	$0100	Generator 8 off
`gen9off`	$0200	Generator 9 off
`gen10off`	$0400	Generator 10 off
`gen11off`	$0800	Generator 11 off
`gen12off`	$1000	Generator 12 off
`gen13off`	$2000	Generator 13 off
`gen14off`	$4000	Generator 14 off

Table 21-6 (continued)
Sound Tool Set constants

Name	Value	Description
Generator status word		
`genAvail`	$0000	Generator available
`ffSynth`	$0100	Free-Form Synthesizer
`noteSynth`	$0200	Note Synthesizer
`lastBlock`	$8000	Last block of wave

Table 21-7
Sound Tool Set data structures

Name	Offset	Type	Definition
SoundParamBlock (sound parameter block)			
waveStart	$00	Pointer	Starting address of wave
waveSize	$04	Word	Waveform size in pages
freqOffset	$06	Word	Output sample rate
docBuffer	$08	Word	DOC buffer start address; low-order byte = 0
bufferSize	$0A	Word	DOC buffer start address; high-order byte = 0
nextWavePtr	$0C	SoundPBPtr	Pointer to start of next wave's parameter block
volSetting	$10	Word	DOC volume setting; high-order byte = 0

Note: The actual assembly-language equates have a lowercase letter *o* in front of all names given in this table.

Table 21-8
Sound Tool Set error codes

Code	Name	Means
$0810	`noDOCFndErr`	No DOC or RAM found
$0811	`docAddrRngErr`	DOC address range error
$0812	`noSAppInitErr`	No SoundStartUp call made
$0813	`invalGenNumErr`	Invalid generator number
$0814	`synthModeErr`	Synthesizer mode error
$0815	`genBusyErr`	Generator already in use
$0817	`mstrIRQNotAssgnErr`	Master IRQ not assigned
$0818	`sndAlreadyStrtErr`	Sound Tool Set already started
$08FF	`unclaimedSndIntErr`	Unclaimed sound interrupt error (reported through System Failure Manager)

Chapter 22

Standard File Operations Tool Set

The **Standard File Operations Tool Set** provides the standard user interface for specifying a file to be opened or saved. The tool set provides dialog boxes that allow the user both to open and save a file on a disk in any drive and to change disks in a drive.

A preview of the Standard File Operations Tool Set routines

To introduce you to the capabilities of the Standard File Operations Tool Set, all of its routines are grouped by function and briefly described in Table 22-1. These routines are described in detail later in this chapter, where they are separated into housekeeping routines (discussed in routine number order) and the rest of the Standard File Operations routines (discussed in alphabetical order).

Table 22-1
Standard File Operations Tool Set routines and their functions

Routine	Description
Housekeeping routines	
SFBootInit	Initializes the Standard File Operations Tool Set; called only by the Tool Locator—must not be called by an application
SFStartUp	Starts up the Standard File Operations Tool Set for use by an application
SFShutDown	Shuts down the Standard File Operations Tool Set
SFVersion	Returns the version number of the Standard File Operations Tool Set
SFReset	Resets the Standard File Operations Tool Set; called only when the system is reset—must not be called by an application
SFStatus	Indicates whether the Standard File Operations Tool Set is active
Other Standard File routines	
SFGetFile	Displays the standard Open File dialog box and returns information about the file selected by the user
SFPutFile	Displays the standard Save File dialog box and returns information about the name of the file to be saved
SFPGetFile	Displays a custom Open File dialog box and returns information about the file selected by the user
SFPPutFile	Displays a custom Save File dialog box and returns information about the name of the file to be saved
SFAllCaps	Allows the application to decide if filenames will be displayed in all uppercase letters or in uppercase and lowercase letters

Standard dialog boxes

The standard Open File dialog box is produced by the SFGetFile routine and is illustrated in Figure 22-1.

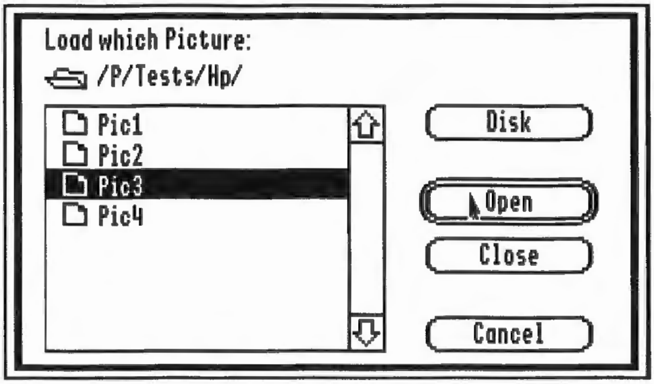


Figure 22-1
Standard Open File dialog box

The standard Save File dialog box is produced by the SFPutFile routine and is illustrated in Figure 22-2.

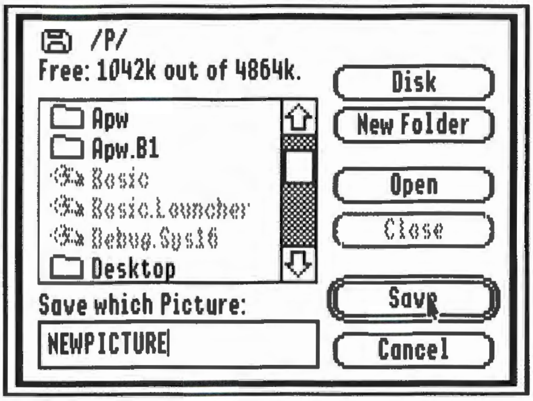


Figure 22-2
Standard Save File dialog box

Standard File dialog templates

The Standard File Operations Tool Set allows you to provide custom dialog boxes for the Open File and Save File dialog boxes. To produce a custom dialog box, you use the SFPPutFile and SFPGetFile routines and provide a pointer to a dialog template in memory. A **dialog template** is a record passed to the Dialog Manager routine GetNewModalDialog. The template contains information about the dialog to be created, including a bounds rectangle and a list of pointers to item templates.

The following sections provide the templates that give the standard Open File and Save File dialog boxes. All of the templates depend on the following strings:

```
Strings          start
SaveStr          str     'Save'
OpenStr          str     'Open'
CloseStr         str     'Close'
DriveStr         str     'Next Drive'
CancelStr        str     'Cancel'
FolderStr        str     'New Folder'
KbFreeStr        str     '^0 free of ^1 K.' ; Dialog Manager routine
                                            ; ParamText replaces ^0 and
                                            ; ^1 with real values from disk
                 end
```

Templates for the standard Open File dialog box

For the Open File dialog box, the item part of the template must include the following items in this exact order:

Item	Item type	ID
Open button	`buttonItem`	1
Close button	`buttonItem`	2
Next button	`buttonItem`	3
Cancel button	`buttonItem`	4
Scroll bar	`scrollBarItem`	5

❖ *Note:* The Standard File dialog box only allows standard scroll bar operations; it does not allow custom scroll bar routines.

Path	`userItem`	6
Files	`userItem`	7
Prompt	`userItem`	8

640 mode

```
GetDialog640    start
                using     Strings

                dc     i'0,0,114,400'
                dc     i'-1'
                dc     i4'0'
                dc     i4'OpenBut640'
                dc     i4'CloseBut640'
                dc     i4'NextBut640'
                dc     i4'CancelBut640'
                dc     i4'Scroll640'
                dc     i4'Path640'
                dc     i4'Files640'
                dc     i4'Prompt640'
                dc     i4'0'
OpenBut640      dc     i'1'
                dc     i'53,265,65,375'
                dc     i'ButtonItem'
                dc     i4'OpenStr'
                dc     i'0'
                dc     i'0'
                dc     i4'0'
CloseBut640     dc     i'2'
                dc     i'71,265,83,375'
                dc     i'ButtonItem'
                dc     i4'CloseStr'
                dc     i'0'
                dc     i'0'
                dc     i4'0'
NextBut640      dc     i'3'
                dc     i'27,265,39,375'
                dc     i'ButtonItem'
                dc     i4'DriveStr'
                dc     i'0'
                dc     i'0'
                dc     i4'0'
CancelBut640    dc     i'4'
                dc     i'97,265,109,375'
                dc     i'ButtonItem'
                dc     i4'CancelStr'
                dc     i'0'
                dc     i'0'
                dc     i4'0'
Scroll640       dc     i'5'
                dc     i'28,214,110,238'
                dc     i'ScrollBarItem'
                dc     i4'0'
                dc     i'0'
                dc     i'3'
                dc     i4'0'
```

```
Path640         dc      i'6'
                dc      i'12,49,24,395'
                dc      i'UserItem'
                dc      i4'0'
                dc      i'0'
                dc      i'0'
                dc      i4'0

Files640        dc      i'7'
                dc      i'28,15,110,215'
                dc      i'UserItem'
                dc      i4'0'
                dc      i'0'
                dc      i'0'
                dc      i4'0'

Prompt640       dc      i'8'
                dc      i'00,15,12,395'
                dc      i'UserItem+$8000'
                dc      i4'0'
                dc      i'0'
                dc      i'0'
                dc      i4'0'

                end
```

320 mode

```
GetDialog320    start
                using   Strings

                dc      i'0,0,114,260'
                dc      i'-1'
                dc      i4'0'
                dc      i4'OpenBut320'
                dc      i4'CloseBut320'
                dc      i4'NextBut320'
                dc      i4'CancelBut320'
                dc      i4'Scroll320'
                dc      i4'Path320'
                dc      i4'Files320'
                dc      i4'Prompt320'
                dc      i4'0'

OpenBut320      dc      i'1'
                dc      i'53,160,65,255'
                dc      i'ButtonItem'
                dc      i4'OpenStr'
                dc      i'0'
                dc      i'0'
                dc      i4'0'
```

```
CloseBut320     dc      i'2'
                dc      i'71,160,83,255'
                dc      i'ButtonItem'
                dc      i4'CloseStr'
                dc      i'0'
                dc      i'0'
                dc      i4'0'

NextBut320      dc      i'3'
                dc      i'27,160,39,255'
                dc      i'ButtonItem'
                dc      i4'DriveStr'
                dc      i'0'
                dc      i'0'
                dc      i4'0'

CancelBut320    dc      i'4'
                dc      i'97,160,109,255'
                dc      i'ButtonItem'
                dc      i4'CancelStr'
                dc      i'0'
                dc      i'0'
                dc      i4'0'

Scroll320       dc      i'5'
                dc      i'27,139,109,151'
                dc      i'ScrollBarItem'
                dc      i4'0'
                dc      i'0'
                dc      i'3'
                dc      i4'0'

Path320         dc      i'6'
                dc      i'14,22,26,256'
                dc      i'UserItem'
                dc      i4'0'
                dc      i'0'
                dc      i'0'
                dc      i4'0'
Files320        dc      i'7'
                dc      i'27,05,109,140'
                dc      i'UserItem'
                dc      i4'0'
                dc      i'0'
                dc      i'0'
                dc      i4'0'
Prompt320       dc      i'8'
                dc      i'00,05,13,255'
                dc      i'UserItem+$8000'
                dc      i4'0'
                dc      i'0'
                dc      i'0'
                dc      i4'0'

                end
```

Templates for the standard Save File dialog box

For the Save File dialog box, the item part of the template must include the following items in this exact order:

Item	Item type	ID
Save button	buttonItem	1
Open button	buttonItem	2
Close button	buttonItem	3
Next button	buttonItem	4
Cancel button	buttonItem	5
Scroll bar	scrollBarItem	6
Path	userItem	7
Files	userItem	8
Prompt	userItem	9
Filename	editItem	10
Free space	statText	11
Create button	button	12

640 mode

```
PutDialog640        start
                    using   Strings

                    dc      i'0,0,120,320'
                    dc      i'-1'
                    dc      i4'0'
                    dc      i4'SaveButP640'
                    dc      i4'OpenButP640'
                    dc      i4'CloseButP640'
                    dc      i4'NextButP640'
                    dc      i4'CancelButP640'
                    dc      i4'ScrollP640'
                    dc      i4'PathP640'
                    dc      i4'FilesP640'
                    dc      i4'PromptP640'
                    dc      i4'EditP640'
                    dc      i4'StatTextP640'
                    dc      i4'CreateButP640'
                    dc      i4'0'

SaveButP640         dc      i'1'
                    dc      i'87,204,99,310'
                    dc      i'ButtonItem'
                    dc      i4'SaveStr'
                    dc      i'0'
                    dc      i'0'
                    dc      i4'0'
```

```
OpenButP640     dc      i'2'
                dc      i'49,204,61,310'
                dc      i'ButtonItem'
                dc      i4'OpenStr'
                dc      i'0'
                dc      i'0'
                dc      i4'0'

CloseButP640    dc      i'3'
                dc      i'64,204,76,310'
                dc      i'ButtonItem'
                dc      i4'CloseStr'
                dc      i'0'
                dc      i'0'
                dc      i4'0'

NextButP640     dc      i'4'
                dc      i'15,204,27,310'
                dc      i'ButtonItem'
                dc      i4'DriveStr'
                dc      i'0'
                dc      i'0'
                dc      i4'0'

CancelButP640   dc      i'5'
                dc      i'104,204,116,310'
                dc      i'ButtonItem'
                dc      i4'CancelStr'
                dc      i'0'
                dc      i'0'
                dc      i4'0'

ScrollP640      dc      i'6'
                dc      i'26,169,88,194'
                dc      i'ScrollBarItem'
                dc      i4'0'
                dc      i'0'
                dc      i'3'
                dc      i4'0'

PathP640        dc      i'7'
                dc      i'00,10,12,315'
                dc      i'UserItem'
                dc      i4'0'
                dc      i'0'
                dc      i'0'
                dc      i4'0'

FilesP640       dc      i'8'
                dc      i'26,10,88,170'
                dc      i'UserItem'
                dc      i4'0'
                dc      i'0'
                dc      i'0'
                dc      i4'0'
```

```
PromptP640      dc      i'9'
                dc      i'88,10,100,200'
                dc      i'UserItem+$8000'
                dc      i4'0'
                dc      i'0'
                dc      i'0'
                dc      i4'0'

EditP640        dc      i'10'
                dc      i'100,10,118,194'
                dc      i'EditLine'
                dc      i4'00'
                dc      i'15'
                dc      i'0'
                dc      i4'0'

StatTextP640    dc      i'11'
                dc      i'12,10,22,200'
                dc      i'StatText+$8000'
                dc      i4'KbFreeStr'
                dc      i'0'
                dc      i'0'
                dc      i4'0'

CreateButP640   dc      i'12'
                dc      i'29,204,41,310'
                dc      i'ButtonItem'
                dc      i4'FolderStr'
                dc      i'0'
                dc      i'0'
                dc      i4'0'

                end
```

320 mode

```
PutDialog320    start
                using   Strings

                dc      i'0,0,128,270'
                dc      i'-1'
                dc      i4'0'
                dc      i4'SaveButP320'
                dc      i4'OpenButP320'
                dc      i4'CloseButP320'
                dc      i4'NextButP320'
                dc      i4'CancelButP320'
                dc      i4'ScrollP320'
                dc      i4'PathP320'
                dc      i4'FilesP320'
                dc      i4'PromptP320'
                dc      i4'EditP320'
                dc      i4'StatTextP320'
                dc      i4'CreateButP320'
                dc      i4'0'
```

```
SaveButP320         dc      i'1'
                    dc      i'93,165,105,265'
                    dc      i'ButtonItem'
                    dc      i4'SaveStr'
                    dc      i'0'
                    dc      i'0'
                    dc      i4'0'

OpenButP320         dc      i'2'
                    dc      i'54,165,66,265'
                    dc      i'ButtonItem'
                    dc      i4'OpenStr'
                    dc      i'0'
                    dc      i'0'
                    dc      i4'0'

CloseButP320        dc      i'3'
                    dc      i'72,165,84,265'
                    dc      i'ButtonItem'
                    dc      i4'CloseStr'
                    dc      i'0'
                    dc      i'0'
                    dc      i4'0'

NextButP320         dc      i'4'
                    dc      i'15,165,27,265'
                    dc      i'ButtonItem'
                    dc      i4'DriveStr'
                    dc      i'0'
                    dc      i'0'
                    dc      i4'0'

CancelButP320       dc      i'5'
                    dc      i'111,165,123,265'
                    dc      i'ButtonItem'
                    dc      i4'CancelStr'
                    dc      i'0'
                    dc      i'0'
                    dc      i4'0'

ScrollP320          dc      i'6'
                    dc      i'26,144,88,157'
                    dc      i'ScrollBarItem'
                    dc      i4'0'
                    dc      i'0'
                    dc      i'3'
                    dc      i4'0'
```

```
PathP320        dc      i'7'
                dc      i'00,10,12,265'
                dc      i'UserItem'
                dc      i4'0'
                dc      i'0'
                dc      i'0'
                dc      i4'0'

FilesP320       dc      i'8'
                dc      i'26,10,88,145'
                dc      i'UserItem'
                dc      i4'0'
                dc      i'0'
                dc      i'0'
                dc      i4'0'

PromptP320      dc      i'9'
                dc      i'88,10,100,170'
                dc      i'UserItem+$8000'
                dc      i4'0'
                dc      i'0'
                dc      i'0'
                dc      i4'0'

EditP320        dc      i'10'
                dc      i'100,10,118,157'
                dc      i'EditLine'
                dc      i4'TitleWin'
                dc      i'15'
                dc      i'0'
                dc      i4'0'

StatTextP320    dc      i'11'
                dc      i'12,10,22,160'
                dc      i'StatText+$8000'
                dc      i4'KbFreeStr'
                dc      i'0'
                dc      i'0'
                dc      i4'0'

CreateButP320   dc      i'12'
                dc      i'33,165,45,265'
                dc      i'ButtonItem'
                dc      i4'FolderStr'
                dc      i'0'
                dc      i'0'
                dc      i4'0'
```

See Chapter 6, "Dialog Manager," in Volume 1 for further details.

The bounding rectangle for the Files user item determines how many files may be displayed at one time. You should set the height of this rectangle to 2 plus 10 times the number of files to show. A height of 122 would allow 12 files to be seen.

Using the Standard File Operations Tool Set

This section discusses how the Standard File Operations Tool Set routines fit into the general flow of an application and gives you an idea of which routines you'll need to use under normal circumstances. Each routine is described in detail later in this chapter.

The Standard File Operations Tool Set depends on the presence of the tool sets shown in Table 22-2 and requires that at least the indicated version of each tool set be present.

Table 22-2
Standard File Operations Tool Set—other tool sets required

Tool set number		Tool set name	Minimum version needed
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0
$03	#03	Miscellaneous Tool Set	1.0
$04	#04	QuickDraw II	1.0
$06	#06	Event Manager	1.0
$0E	#14	Window Manager	1.3
$10	#16	Control Manager	1.3
$0F	#15	Menu Manager	1.3
$14	#20	LineEdit Tool Set	1.0
$15	#21	Dialog Manager	1.1

To use the Standard File Operations Tool Set routines, your application must call the SFStartUp routine before making any other Standard File calls. To save memory space, you can choose to have your application make the SFStartUp call only when it needs to present the dialog boxes. Use the LoadOneTool routine in the Tool Locator if you wish to use this method. See the section "SFStartUp" in this chapter for an example.

Use the SFShutDown routine to shut down the Standard File Operations Tool Set after you have finished using it. If you wish, you can unload the tool set with the Tool Locator routine UnLoadOneTool, which will unload the tool set from memory and thus save space.

Important

If you choose to unload the Standard File Operations Tool Set, be sure to reload it with a LoadOneTool call before making the SFStartUp call again.

When the user makes a request to open a file, use the SFGetFile routine in your application to present the standard Open File dialog box and retrieve the file name. SFGetFile allows you to specify where the standard dialog box will be placed on the screen, specify the prompt at the top of the box, and filter the type of files to be displayed in the box. The routine does not allow you to modify the appearance of the box; if you wish to construct your own custom dialog box, use the SFPGetFile routine.

When the user makes a request to save a file, use the SFPutFile routine to present the standard dialog Save File dialog box. SFPutFile allows you to specify where the standard dialog box will be placed on the screen, the prompt at the top of the box, and the maximum number of characters the user may type.

Like SFGetFile, SFPutFile does not allow you to modify the appearance of the box; if you wish to construct your own custom dialog box, use the SFPPutFile routine.

$0117 SFBootInit

Initializes Standard File Operations Tool Set; called only by the Tool Locator.

Warning

An application must never make this call.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C Call must not be made by an application.

$0217 SFStartUp

Starts up the Standard File Operations Tool Set for use by an application.

Important

Your application must make this call before it makes any other Standard File calls.

You may choose to have your application call SFStartUp only when it is needed, thus freeing memory for other uses. The number of the Standard File Operations Tool Set is $17, so a typical sequence of calls (in a pseudocode format) might be

LoadOneTool ($17,$0101)
SFStartUp (appropriate parameters)
SFGetFile (appropriate parameters)
SFShutDown
UnloadOneTool ($17)

Parameters

Stack before call

previous contents	
userID	**Word**—ID number of application
dPageAddr	**Word**—Bank $0 starting address for one page of direct-page space
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SFStartUp(userID,dPageAddr)

Word      userID;

Word      dPageAddr;
```

$0317 SFShutDown

Shuts down the Standard File Operations Tool Set. Your application may call SFShutDown immediately after Standard File Operations are completed, thus freeing memory for other uses.

Important

If your application has started up Standard File Operations, the application must make this call before it quits.

The number of the Standard File Operations Tool Set is $17, so a typical sequence of calls (in a pseudocode format) might be

LoadOneTool ($17,$0101)
SFStartUp (appropriate parameters)
SFGetFile (appropriate parameters)
SFShutDown
UnloadOneTool ($17)

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void SFShutDown()
```

$0417 SFVersion

Returns the version number of the Standard File Operations Tool Set.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
versionInfo	**Word**—Version number of Standard File Operations Tool Set
	← **SP**

Errors None

C

```
extern pascal Word SFVersion()
```

$0517 SFReset

Resets the Standard File Operations Tool Set; called only when the system is reset.

Warning
An application must never make this call.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C Call must not be made by an application.

$0617 SFStatus

Indicates whether the Standard File Operations Tool Set is active.

SFStatus returns TRUE if SFStartUp has been called and SFShutDown has not been called. The routine returns FALSE if SFStartUp has not been called at all or if SFShutDown has been called since the last time SFStartUp was called.

Parameters

Stack before call

Stack	
previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

Stack	
previous contents	
activeFlag	**Word**—BOOLEAN; TRUE if Standard File active, FALSE if inactive
	← **SP**

Errors

None

C

```
extern pascal Boolean SFStatus()
```

$0D17 SFAllCaps

Allows an application to decide if filenames will be displayed in all uppercase letters or in uppercase and lowercase letters. If *allCapsFlag* is set to FALSE, the initial letter of the filename will be uppercase and the first letter after a period will be uppercase; all other letters will be lowercase.

Parameters

Stack before call

previous contents
allCapsFlag

allCapsFlag: **Word**—BOOLEAN; TRUE for uppercase filenames, FALSE for mixed

← **SP**

Stack after call

previous contents

← **SP**

Errors

None

C

```
extern pascal void SFAllCaps(allCapsFlag)

Boolean     allCapsFlag;
```

$0917 SFGetFile

Displays the standard Open File dialog box and returns information about the file selected by the user.

Parameters

Stack before call

previous contents	
whereX	**Word**—INTEGER; X coordinate of upper left corner of dialog box
whereY	**Word**—INTEGER; Y coordinate of upper left corner of dialog box
-- *promptPtr* --	**Long**—POINTER to string to display at top of dialog box
-- *filterProcPtr* --	**Long**—POINTER to filter procedure; NIL for none
-- *typeListPtr* --	**Long**—POINTER to typelist record; NIL to display all files
-- *replyPtr* --	**Long**—POINTER to a reply record
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SFGetFile(whereX,whereY,promptPtr,
filterProcPtr,typeListPtr,replyPtr)
Integer     whereX;
Integer     whereY;
Pointer     promptPtr;
WordProcPtr     filterProcPtr;
Pointer     typeListPtr;
SFReplyRecPtr     replyPtr;
```

You can also use the following alternate form of the call:

```
extern pascal void SFGetFile(where,promptPtr,
filterProcPtr,typeListPtr,replyPtr)

Point       where;

Pointer     promptPtr;

WordProcPtr     filterProcPtr;

Pointer     typeListPtr;

SFReplyRecPtr     replyPtr;
```

More about parameters

The filter procedure pointed to by *filterProcPtr* determines which files will be displayed in the dialog box. Set *filterProcPtr* to NIL to prevent the procedure from being called. The filter procedure is called in full native mode, in the same way as one would call a Pascal function having one long parameter.

The calling sequence inside SFGetFile is as follows:

```
PushWord #0           ; Space for result

PushLong #DirEntry    ; Pointer to directory entry ($27 bytes long)

jsl FilterProc

PopWord Result
```

The procedure must strip the four bytes of the directory entry off the stack and return with the result at the top of the stack. The result indicates what the procedure wants to do with the file, as shown in Table 22-3.

Table 22-3
Filter procedure results

Value	Name	Description
0	`noDisplay`	Don't display file
1	`noSelect`	Display file but don't allow user to select the file
2	`displaySelect`	Display file and allow user to select the file

The directory entry for the file is illustrated in Figure 22-3.

Offset		Field
$0	storage_type / name_length	1 byte
1		
2		
	file_name	15 bytes
0F		
10	file_type	1 byte
11	key_pointer	2 bytes
12		
13	blocks_used	2 bytes
14		
15		
16	EOF	3 bytes
17		
18		
19	creation date and time	4 bytes
1A		
1B		
1C	version	1 byte
1D	min_version	1 byte
1E	access	1 byte
1F	aux_type	2 bytes
20		
21		
22	modification date and time	4 bytes
23		
24		
25	header_pointer	2 bytes
26		

Figure 22-3
File directory entry

For more information on the fields of the directory entry, see the *Apple IIGS ProDOS 16 Reference.*

The *typeListPtr* parameter points to a record containing a list of file types to display (set the pointer to NIL to display all file types). The list has the form shown in Figure 22-4.

Offset	Field	
$0	numEntries	**Byte—**Total number of entries in list
1	fileType1	**Byte—**First file type to display
2	fileType2	**Byte—**Second file type to display
	lastfileType	**Byte—**Last file type to display

Figure 22-4
Typelist record

(continued)

If you specify both a *filterProcPtr* and a *typeListPtr*, only files of the right file type or types are passed on to the filterProc procedure.

The *replyPtr* parameter points to a reply record that has the form shown in Figure 22-5.

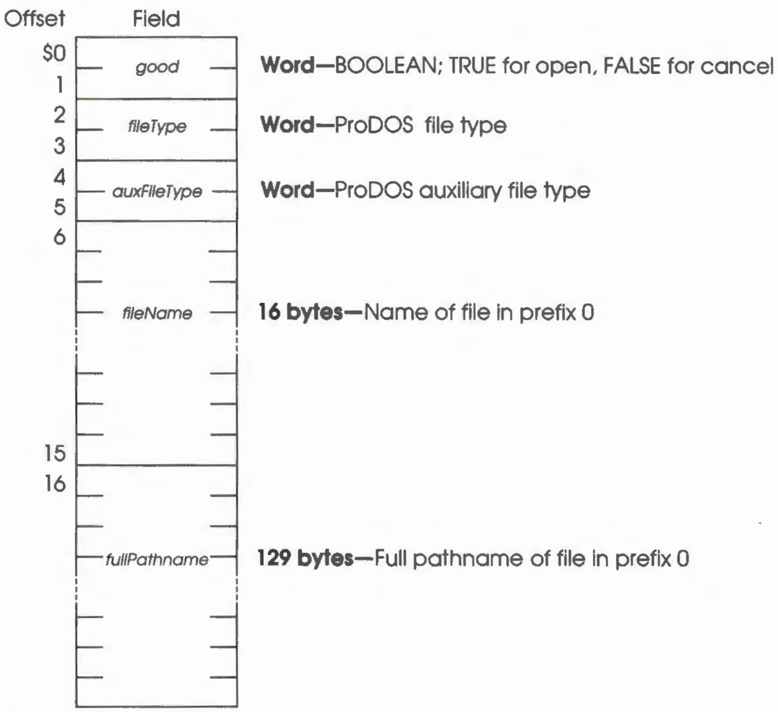


Figure 22-5
Reply record

When the dialog box is displayed, the files from prefix 0 are shown. When the file is selected, the name of the file is returned in the record pointed to by *replyPtr* and prefix 0 is set to the directory containing the selected file. If the user cancels the operation, prefix 0 is left at whatever directory is being shown at the time.

❖ *Macintosh programmers:* The Disk button works differently from the Drive button in the Macintosh. When a user clicks on the Disk button, Standard File first looks at the disk in the drive the current disk is expected to be in. If the current disk is no longer in that drive, the disk in that drive becomes the current disk. If the current disk is still there, the Disk button moves to the next disk in the ProDOS chain. The Disk button works this way because a user can change disks without the system knowing about it.

$0B17 SFPGetFile

Displays a custom Open File dialog box and returns information about the file selected by the user.

Parameters

Stack before call

previous contents	
whereX	**Word**—INTEGER; X coordinate of upper left corner of dialog box
whereY	**Word**—INTEGER; Y coordinate of upper left corner of dialog box
— *promptPtr* —	**Long**—POINTER to string to display at top of dialog box
— *filterProcPtr* —	**Long**—POINTER to filter procedure; NIL for none
— *typeListPtr* —	**Long**—POINTER to a typelist record; NIL to display all files
— *dlgTempPtr* —	**Long**—POINTER to a dialog template in memory
— *dialogHookPtr* —	**Long**—POINTER to a routine called every time ModalDialog returns a hit
— *replyPtr* —	**Long**—POINTER to a reply record
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SFPGetFile(whereX,whereY,promptPtr,filterProcPtr,
typeListPtr,dlgTempPtr,dialogHookPtr,replyPtr)

Integer     whereX;

Integer     whereY;

Pointer     promptPtr;

WordProcPtr     filterProcPtr;

Pointer     typeListPtr;

DlgTempPtr     dlgTempPtr;
```

```
VoidProcPtr     dialogHookPtr;
SFReplyRecPtr     replyPtr;
```

The dlgTempPtr and dialogHookPtr parameters

SFPGetFile works like SFGetFile, except for the addition of the *dlgTempPtr* and *dialogHookPtr* parameters. The *dlgTempPtr* provides a pointer to a dialog template; the template is a record passed to the Dialog Manager routine GetNewModalDialog. The template contains information about the dialog to be created, including a bounds rectangle and a list of pointers to item templates.

For the Open File dialog box, the item part of the template must include the following items in this exact order:

Item	Item type	ID
Open button	buttonItem	1
Close button	buttonItem	2
Next button	buttonItem	3
Cancel button	buttonItem	4
Scroll bar	scrollbarItem	5

❖ *Note:* The Standard File dialog box only allows standard scroll bar operations; it does not allow custom scroll bar routines.

Path	userItem	6
Files	userItem	7
Prompt	userItem	8

For more information and examples of dialog templates, see the section “Standard File Dialog Templates” in this chapter.

The *dialogHookPtr* parameter is a pointer to the routine called by SFPGetFile every time ModalDialog returns an item hit. The routine is passed a pointer to the dialog port and a pointer to the item-hit word. If the dialogHook routine wants to handle the item hit, it should handle it and set the hit to 0. If it wants SFPGetFile to handle the item hit, it should leave it unchanged.

The routine is called as follows:

```
PushLong #DialogPort
PushLong #ItemHit
jsl DialogHook
lda ItemHit
```

Your routine must be certain to strip the two longs (eight bytes) representing the DialogPort and ItemHit pointers off the stack before returning to SFPGetFile.

$0C17 SFPPutFile

Displays a custom Save File dialog box and returns information about the name of the file to be saved.

Parameters

Stack before call

previous contents	
whereX	**Word**—INTEGER; X coordinate of upper left corner of dialog box
whereY	**Word**—INTEGER; Y coordinate of upper left corner of dialog box
-- *promptPtr* --	**Long**—POINTER to string to display at top of dialog box
-- *origNamePtr* --	**Long**—POINTER to string holding name that appears as default
maxLen	**Word**—INTEGER; maximum number of characters user may type
-- *dlgTempPtr* --	**Long**—POINTER to a dialog template in memory
-- *dialogHookPtr* --	**Long**—POINTER to a routine called every time ModalDialog returns a hit
-- *replyPtr* --	**Long**—POINTER to a reply record
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SFPPutFile(whereX,whereY,promptPtr,origNamePtr,maxLen,
dlgTempPtr,dialogHookPtr,replyPtr)

Integer      whereX;
Integer      whereY;
Pointer      promptPtr;
Pointer      origNamePtr;
unsigned int      maxLen;
DlgTempPtr     dlgTempPtr;
VoidProcPtr     dialogHookPtr;
SFReplyRecPtr     replyPtr;
```

You can also use the following alternate form of the call:

```
extern pascal void SFPPutFile(where,promptPtr,origNamePtr,maxLen,dlgTempPtr,
dialogHookPtr,replyPtr)

Point     where;
Pointer      promptPtr;
Pointer      origNamePtr;
unsigned int      maxLen;
DlgTempPtr     dlgTempPtr;
VoidProcPtr     dialogHookPtr;
SFReplyRecPtr     replyPtr;
```

The dlgTempPtr and dialogHookPtr parameters

This routine works like SFPutFile, except for the addition of the *dlgTempPtr* and *dialogHookPtr* parameters. The *dlgTempPtr* provides a pointer to a dialog template; the template is a record passed to the Dialog Manager routine GetNewModalDialog. The template contains information about the dialog to be created, including a bounds rectangle and a list of pointers to item templates.

The item part of the template must include the following items in this exact order:

Item	Item type	ID
Save button	buttonItem	1
Open button	buttonItem	2
Close button	buttonItem	3
Next button	buttonItem	4
Cancel button	buttonItem	5
Scroll bar	scrollBarItem	6
Path	userItem	7
Files	userItem	8
Prompt	userItem	9
Filename	editItem	10
Free space	statText	11
Create button	button	12

For more information and examples of dialog templates, see the section "Standard File Dialog Templates" in this chapter and Chapter 6, "Dialog Manager," in Volume 1 for further details.

The bounding rectangle for the Files user item determines how many files may be displayed. You should set the height of this rectangle to 2 plus 10 times the number of files to show at one time. A height of 122 would allow 12 files to be seen at one time.

The *dialogHookPtr* parameter is a pointer to the routine called by SFPPutFile every time ModalDialog returns an item hit. The routine is passed a pointer to the dialog port and a pointer to the item-hit word. If the DialogHook routine wants to handle the item hit, it should handle it and set the hit to 0. If the DialogHook routine wants SFPPutFile to handle the item hit, it should leave the hit unchanged.

The routine is called as follows in assembly language:

```
PushLong #DialogPort

PushLong #ItemHit

jsl DialogHook

lda ItemHit
```

Your routine must be certain to strip the two longs (eight bytes) representing the DialogPort and the ItemHit off the stack before returning to SFPPutFile.

$0A17 SFPutFile

Displays the standard Save File dialog box and returns information about the name of the file to be saved.

Parameters

Stack before call

previous contents	
whereX	**Word**—INTEGER; X coordinate of upper left corner of dialog box
whereY	**Word**—INTEGER; Y coordinate of upper left corner of dialog box
— *promptPtr* —	**Long**—POINTER to string to display at top of dialog box
— *origNamePtr* —	**Long**—POINTER to string holding name that appears as default
maxLen	**Word**—INTEGER; maximum number of characters user may type
— *replyPtr* —	**Long**—POINTER to a reply record
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SFPutFile(whereX,whereY,promptPtr,
origNamePtr,maxLen,replyPtr)

Integer     whereX;
Integer     whereY;
Pointer     promptPtr;
Pointer     origNamePtr;
unsigned int     maxLen;
SFReplyRecPtr     replyPtr;
```

You can also use the following alternate form of the call:

```
extern pascal void SFPutFile(where,promptPtr,origNamePtr,maxLen,replyPtr)
Point      where;
Pointer     promptPtr;
Pointer     origNamePtr;
unsigned int     maxLen;
SFReplyRecPtr     replyPtr;
```

More about parameters

The *maxLen* parameter specifies the maximum number of characters a user may type. Most applications will use 15 for this value, but if the application wants to add a suffix to the file name, the *maxLen* value must be shortened. Values greater than 15 are not valid.

The *replyPtr* points to the same type of reply record as that of SFGetFile. See Figure 22-5 in the section "SFGetFile" in this chapter.

When the dialog box is first displayed, all the files from prefix 0 are shown. This lets the user know what names are in use and prevents use of a name that already exists. The user can open any directories shown as selectable; files not shown as selectable cannot be opened. When the user clicks on the Save button or presses the Return key, the name of the file is returned in the reply record and prefix 0 is set to the directory containing the selected file. If the user cancels the operation, prefix 0 is restored to its original state.

SFPutFile also checks to see if the file already exists. If it does, SFPutFile displays a dialog asking if is OK to destroy the existing file.

If the user cancels the operation, prefix 0 is left at whatever directory is being shown at the time.

❖ *Macintosh programmers:* The Disk button works differently from the Drive button in the Macintosh. When a user clicks on the Disk button, Standard File first looks at the disk in the drive the current disk is expected to be in. If the current disk is no longer in that drive, the disk in that drive becomes the current disk. If the current disk is still there, the Disk button moves to the next disk in the ProDOS chain. The Disk button works this way because a user can change disks without the system knowing about it.

Standard File Operations Tool Set summary

This section briefly summarizes the constants and data structures contained in the Standard File Operations Tool Set. There are no tool set error codes for the Standard File Operations Tool Set.

Important

These definitions are provided in the appropriate interface file.

Table 22-4
Standard File Operations Tool Set constants

Name	Value	Description
Filter procedure results		
`noDisplay`	$0000	Don't display file
`noSelect`	$0001	Display file, but don't allow user to select it
`displaySelect`	$0002	Display file and allow user to select it

Table 22-5
Standard File Operations Tool Set data structures

Name	Offset	Type	Definition
Reply record			
good	$00	Boolean	TRUE for Open, FALSE for Cancel
fileType	$02	Word	ProDOS file type
auxFileType	$04	Word	ProDOS aux file type
fileName	$06	16 bytes	Name of the file in prefix .0
fullPathname	$16	129 bytes	Full pathname of selected file

Note: The actual assembly-language equates have a lowercase letter *o* in front of all of the names given in this table.

Chapter 23

Text Tool Set

The **Text Tool Set** provides an interface between Apple II character device drivers, which must be executed in emulation mode, and new applications running in native mode. It also provides a means of redirection of I/O through RAM-based drivers. The Text Tool Set makes it possible to deal with the text screen without switching 65816 processor modes and moving to bank zero. Dispatches to RAM-based drivers will occur in full native mode.

A preview of the Text Tool Set routines

To introduce you to the capabilities of the Text Tool Set, all of its routines are grouped by function and briefly described in Table 23-1. These routines are described in detail later in this chapter, where they are separated into housekeeping routines (discussed in routine number order) and the rest of the Text Tool Set routines (discussed in alphabetical order).

Table 23-1
Text Tool Set routines and their functions

Routine	Description
Housekeeping routines	
TextBootInit	Initializes the Text Tool Set; called only by the Tool Locator—must not be called by an application
TextStartUp	Starts up the Text Tool Set for use by an application
TextShutDown	Shuts down the Text Tool Set when an application quits
TextVersion	Returns the version number of the Text Tool Set
TextReset	Resets the Text Tool Set; called only when the system is reset—must not be called by an application
TextStatus	Indicates whether the Text Tool Set is active

(continued)

Table 23-1 (continued)
Text Tool Set routines and their functions

Routine	Description
Text global functions	
SetInGlobals	Sets the global parameters for the input device
SetOutGlobals	Sets the global parameters for the output device
SetErrGlobals	Sets the global parameters for the error output device
GetInGlobals	Returns the current values for the input device global parameters
GetOutGlobals	Returns the current values for the output device global parameters
GetErrGlobals	Returns the current values for the error output device global parameters
I/O directing routines	
SetInputDevice	Sets the input device to a specified type and location
SetOutputDevice	Sets the output device to a specified type and location
SetErrorDevice	Sets the error output device to a specified type and location
GetInputDevice	Returns the type of driver installed as the input device
GetOutputDevice	Returns the type of driver installed as the output device
GetErrorDevice	Returns the type of driver installed as the error output device
Text routines	
InitTextDev	Initializes a specified text device
CtlTextDev	Passes a control code to a specified text device
StatusTextDev	Executes a status call to a specified text device
WriteChar	Combines a specified character with the output global AND mask and OR mask and writes the character to the output text device
ErrWriteChar	Combines a specified character with the output global AND mask and OR mask and writes the character to the error output text device
WriteLine	Combines a pointed-to Pascal-type string (first byte of string specifies length) with the output global masks, concatenates a carriage return (for BASIC or RAM-based drivers) or a carriage return and line feed (for Pascal drivers), and then writes the string to the output text device
ErrWriteLine	Combines a pointed-to Pascal-type string (first byte of string specifies length) with the output global masks and then writes the string to the error output text device
WriteString	Combines a pointed-to Pascal-type string (first byte of string specifies length) with the output global masks and then writes the string to the output text device
ErrWriteString	Combines a pointed-to Pascal-type string (first byte of string specifies length) with the output global masks and then writes the string to the error output text device
TextWriteBlock	Combines a specified character string with the output global masks and then writes the string to the output text device
ErrWriteBlock	Combines a specified character string with the output global masks and then writes the string to the error output text device
WriteCString	Combines a pointed-to C string (string terminates with $00) with the output global masks and then writes the string to the output text device
ErrWriteCString	Combines a pointed-to C string (string terminates with $00) with the output global masks and then writes the string to the error output text device

Table 23-1 (continued)
Text Tool Set routines and their functions

Routine	Description
ReadChar	Reads a character obtained from the input text device, combines it with the input global masks, and returns the character on the stack
ReadLine	Reads a character string from the input text device, combines it with the input global masks, and writes the string to a specified memory location
TextReadBlock	Reads a block of characters from the input text device, combines it with the input global masks, and writes the block to a specified memory location

Using the I/O directing routines

The I/O directing routines direct I/O from the Text Tool Set to a specific type of character device driver or get information about the directing of a specific I/O driver. The types of character device drivers are listed in Table 23-2.

Table 23-2
Character device driver types

Type	Description
0	BASIC device driver
1	Pascal device driver
2	RAM-based device driver
≥3	Illegal driver type

BASIC device drivers must support the standard Apple II BASIC device driver entry points (INIT, INPUT, and OUTPUT).

❖ *Note:* BASIC devices use the Apple II I/O hooks ($36–$39) in absolute zero page.

Any desk accessories using a Text Tool BASIC device driver should save and restore the global masks, device descriptors, and I/O hooks when entering or leaving the desk accessory.

Pascal device drivers must support the standard Apple II Pascal 1.1 device driver entry points (INIT, READ, WRITE, and status). The optional Pascal 1.1 control entry point is supported by the Text Tool Set but does not necessarily have to be supported by the device. The application must find out if the device supports the optional Pascal entry points.

RAM-based device drivers may be located at any address and in any bank. A RAM-based driver must support five entry points, as follows:

RAMDRIVER Base Address	INIT (initialization) entry point
RAMDRIVER Base Address + 3	READ entry point
RAMDRIVER Base Address + 6	WRITE entry point
RAMDRIVER Base Address + 9	STATUS entry point
RAMDRIVER Base Address + 12	CONTROL entry point

RAM-based drivers are called by the Text Tool Set in native mode (m and x bits set to 16 bits) and should return to the Text Tools via an RTL instruction. The Text Tools pass data or ASCII characters to the RAM-based driver via the low-order byte of the 16-bit accumulator.

A RAM-based driver should not make any assumptions about the state of the data bank register or the direct page register. I/O performed by RAM-based drivers should operate on a single-character basis. The Text Tool Set, not the RAM-based driver, manipulates the text with regard to the global masks before passing the text to the RAM-based driver. All Text Tool Set routines that interface to different string types are supported by the tool set and not the device.

Using the text routines

The following example demonstrates how to set up global parameters, initialize the text devices, send output to the error and output devices, and read input from the input device.

```
         list           on

         instime        off

         absaddr        on

         gen            off

         symbol         off

         keep           testit

         65816          on

         msb            on

         mcopy          mm.macros

         mcopy          misc.macros

         mcopy          tt.macros

         test           start
```

```
C2 30   rep           #$30
        longa         on
        longi         on
        _TextStartUp
        bcc           NoErr0           ; If no error
        jmp           ToolError        ; else
NoErr0  anop
        pea           $00FF            ; AND mask
        pea           $0080            ; OR mask
        _SetInGlobals
        bcc           NoErr1           ; If no error
        jmp           ToolError        ; else
NoErr1  anop
        pea           $00FF            ; AND mask
        pea           $0080            ; OR mask
        _SetOutGlobals
        bcc           NoErr2           ; If no error
        jmp           ToolError        ; else
NoErr2  anop
        pea           $00FF            ; AND mask
        pea           $0080            ; OR mask
        _SetErrGlobals
        bcc           NoErr3           ; If no error
        jmp           ToolError        ; else
NoErr3  anop
        pea           $0000            ; 0 = basic driver
        pea           $0000            ; in slot #3
        pea           $0003            ;
        _SetInputDevice
        bcc           NoErr4           ; If no error
        jmp           ToolError        ; else
```

```
NoErr4  anop
        pea         $0000           ; 0 = basic driver
        pea         $0000           ; in slot #3
        pea         $0003           ;
        _SetOutputDevice
        bcc         NoErr5          ; If no error
        jmp         ToolError       ; else
NoErr5  anop
        pea         $0000           ; 0 = basic driver
        pea         $0000           ; in slot #1
        pea         $0001           ;
        _SetErrorDevice
        bcc         NoErr6          ; If no error
        jmp         ToolError       ; else
NoErr6  anop
        pea         $0000           ; Initialize the input device
        _InitTextDev
        bcc         NoErr7          ; If no error
        jmp         ToolError       ; else
NoErr7  anop
        pea         $0001           ; Initialize the output device
        _InitTextDev
        bcc         NoErr8          ; If no error
        jmp         ToolError       ; else
NoErr8  anop
        pea         $0002           ; Initialize the error device
        _InitTextDev
        bcc         NoErr9          ; If no error
        jmp         ToolError       ; else
```

```
NoErr9   anop
         pha                          ; Space for result
         pea        $0000             ; Don't echo
         _ReadChar                    ; Wait for any key
         bcc        NoErr10           ; If no error
         jmp        ToolError         ; else
NoErr10  anop
         pushlong   #buffer           ; Pointer to input buffer
         pea        $0080             ; Offset into buffer
         pea        $0100             ; 256 characters is max block size
         pea        $0001             ; Echo characters to output device
         _ReadBlock                   ; Wait for any key
         bcc        NoErr11           ; If no error
         jmp        ToolError         ; else
NoErr11  anop
         pushlong   #outstring        ; Pointer to the output device string
         _WriteCString                ; and send it out
         bcc        NoErr12           ; If no error
         jmp        ToolError         ; else
NoErr12  anop
         pushlong   #ErrString        ; Pointer to the error device string
         _ErrWriteCString             ; and send it out
         bcc        NoErr13           ; If no error
         jmp        ToolError         ; else
NoErr13  anop
         rts
```

```
ToolError anop
         pha                                  ; Error code for display
         pushlong      #FailMsg               ; Pointer to error message
         _SysFailMgr
         outstring     anop
         dc            c'This string should go on the screen'
         dc            h'0D'                  ; A carriage return (basic = auto lf)
         dc            h'00'
         errstring     anop
         dc            c'This string should go on the paper'
         dc            h'0D'                  ; A carriage return (basic = auto lf)
         dc            h'00'
         msb           off
FailMsg  anop
         dc            h'22'                  ; Character count
         dc            c'A tool call returned with error = '
         end
```

Using the Text Tool Set

This section discusses how the Text Tool Set routines fit into the general flow of an application and gives you an idea of which routines you'll need to use under normal circumstances. Each routine is described in detail later in this chapter.

The Text Tool Set depends on the presence of the tool sets shown in Table 23-3 and requires that at least the indicated version of the tool set be present.

Table 23-3
Text Tool Set—other tool sets required

Tool set number		Tool set name	Minimum version needed
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0

Your application should make a TextStartUp call before making any other Text Tool Set calls.

❖ *Note:* At the time of publication, the TextStartUp call was not an absolute requirement because the Tool Locator was automatically starting up the Text Tool Set at boot time. However, you should make the call anyway to guarantee that your application remains compatible with all future versions of the system.

Your application should also make the TextShutDown call when the application quits.

The Text Tool Set has global routines that are used to set or read the current global parameters used by RAM and the Pascal and BASIC text tools. The tool set also has I/O directing routines that direct I/O from the tool set to a specific type of character device driver or get information about directing a specific I/O driver. Finally, the tool set has text routines that interface with any BASIC, Pascal 1.1, or RAM-based character device driver.

$010C TextBootInit

Initializes the Text Tool Set; called only by the Tool Locator.

Warning

An application must never make this call.

TextBootInit sets up the default device parameters as follows:

- Input device type is BASIC.
- Output device type is BASIC.
- Error output device type is BASIC.
- Input device resides in slot #3.
- Output device resides in slot #3.
- Error output device resides in slot #3.
- Global input AND mask is set to $FF.
- Global input OR mask is set to $80.
- Global output AND mask is set to $FF.
- Global output OR mask is set to $80.
- Global error output AND mask is set to $FF.
- Global error output OR mask is set to $80.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C Call must not be made by an application.

$020C TextStartUp

Starts up the Text Tool Set for use by an application. Your application should make a TextStartUp call before making any other Text Tool Set calls.

❖ Note: At the time of publication, the TextStartUp call was not an absolute requirement because the Tool Locator was automatically starting up the Text Tool Set at boot time. However, you should make the call anyway to guarantee that your application remains compatible with all future versions of the system.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void TextStartUp()
```

$030C TextShutDown

Shuts down the Text Tool Set when an application quits.

Important

If your application has started up the Text Tool Set, the application must make this call before it quits.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void TextShutDown()
```

$040C TextVersion

Returns the version number of the Text Tool Set.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
versionInfo	**Word**—Version number of Text Tool Set
	← **SP**

Errors

None

C

```
extern pascal Word TextVersion()
```

$050C TextReset

Resets the Text Tool Set; called only when the system is reset.

Warning
An application must never make this call.

Resets the device parameters to the default values as follows:

- Input device type is BASIC.
- Output device type is BASIC.
- Error output device type is BASIC.
- Input device resides in slot #3.
- Output device resides in slot #3.
- Error output device resides in slot #3.
- Global input AND mask is set to $FF.
- Global input OR mask is set to $80.
- Global output AND mask is set to $FF.
- Global output OR mask is set to $80.
- Global error output AND mask is set to $FF.
- Global error output OR mask is set to $80.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C Call must not be made by an application.

$060C TextStatus

Indicates whether the Text Tool Set is active.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
activeFlag	**Word**—BOOLEAN; TRUE if Text Tool Set active, FALSE if inactive
	← **SP**

Errors

None

C

```
extern pascal Boolean TextStatus()
```

$160C CtlTextDev

Passes a control code to a specified text device. The control codes passed depend on the device used and are outside the scope of this manual.

For Pascal device drivers, this is an optional entry point that may not be supported by a particular Pascal device. Because BASIC devices do not support this routine, an error occurs if this call is made to a BASIC device.

Parameters

Stack before call

previous contents	
deviceNum	**Word**—Device to control (0 = input, 1 = output, 2 = error output)
controlCode	**Word**—Control code (in low-order byte) to pass to device
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$0C01	`badDevType`	Illegal device type

❖ *Note:* The remaining errors should occur only for Pascal devices.

$0C02	`badDevNum`	Illegal device number
$0C03	`badMode`	Illegal operation
$0C04	`unDefHW`	Undefined hardware error
$0C05	`lostDev`	Lost device: device no longer on-line
$0C06	`lostFile`	File no longer in diskette directory
$0C07	`badTitle`	Illegal filename
$0C08	`noRoom`	Insufficient space on specified diskette
$0C09	`noDevice`	Specified volume not on-line
$0C0A	`noFile`	Specified file not in directory of specified volume
$0C0B	`dupFile`	Duplicate file: attempt to rewrite a file when a file of that name already exists
$0C0C	`notClosed`	Attempt to open file that is already open

$0C0D	`notOpen`	Attempt to access a closed file
$0C0E	`badFormat`	Error in reading real or integer number
$0C0F	`ringBuffOFlo`	Ring buffer overflow: characters arriving faster than the input buffer can accept them
$0C10	`writeProtected`	Specified diskette is write protected
$0C40	`devErr`	Device error: device failed to complete a read or write correctly

C

```
extern pascal void CtlTextDev(deviceNum,controlCode)
Word     deviceNum;
Word     controlCode;
```

$1F0C ErrWriteBlock

Combines a specified character string with the output global masks and then writes the string to the error output text device. The string is specified by the parameters *textPtr* + *offset*, with the length specified by the *count* parameter.

Parameters

Stack before call

previous contents	
-- *textPtr* --	**Long**—POINTER to start of ASCII text
offset	**Word**—Offset within ASCII text
count	**Word**—String length
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Pascal device errors	Any of the errors $0C02–$0C40 as listed in the section "Text Tool Set Summary" at the end of this chapter

C

```
extern pascal void ErrWriteBlock(textPtr,offset,count)

Pointer     textPtr;

Word      offset;

Word      count;
```

$190C ErrWriteChar

Combines a specified character with the output global AND mask and OR mask and writes the character to the error output text device.

Parameters

Stack before call

previous contents	
theChar	**Word**—Character to write (in low-order byte)
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Pascal device errors — Any of the errors $0C02–$0C40 as listed in the section "Text Tool Set Summary" at the end of this chapter

C

```
extern pascal void ErrWriteChar(theChar)

Word     theChar;
```

$210C ErrWriteCString

Combines a pointed-to C-type string (string terminates with $00) with the output global masks and then writes the string to the error output text device.

Parameters

Stack before call

previous contents
-- *cStrPtr* --

cStrPtr: **Long**—POINTER to C-type string to be written

← **SP**

Stack after call

previous contents

← **SP**

Errors

Pascal device errors	Any of the errors $0C02–$0C40 as listed in the section "Text Tool Set Summary" at the end of this chapter

C

```
extern pascal void ErrWriteCString(cStrPtr)

Pointer     cStrPtr;
```

$1B0C ErrWriteLine

Combines a pointed-to Pascal-type string (first byte of string specifies length) with the output global masks and then writes the string to the error output text device. For BASIC and RAM-based drivers, the routine concatenates a carriage return to the string. For Pascal drivers, the routine concatenates a carriage return and a line feed to the string.

Parameters

Stack before call

previous contents
— *strPtr* —	**Long**—POINTER to Pascal-type string

← **SP**

Stack after call

previous contents

← **SP**

Errors

Pascal device errors — Any of the errors $0C02–$0C40 as listed in the section "Text Tool Set Summary" at the end of this chapter

C

```
extern pascal void ErrWriteLine(strPtr)

Pointer     strPtr;
```

$1D0C ErrWriteString

Combines a pointed-to Pascal-type string (first byte of string specifies length) with the output global masks and then writes the string to the error output text device.

Parameters

Stack before call

previous contents
-- *strPtr* --

strPtr: **Long**—POINTER to Pascal-type string to be written

← **SP**

Stack after call

previous contents

← **SP**

Errors

Pascal device errors — Any of the errors $0C02–$0C40 as listed in the section "Text Tool Set Summary" at the end of this chapter

C

```
extern pascal void ErrWriteString(strPtr)

Pointer     strPtr;
```

$0E0C GetErrGlobals

Returns the current values for the error output device global parameters.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
andMask	**Word**—Global AND mask for error output device (low-order byte only)
orMask	**Word**—Global OR mask for error output device (low-order byte only)
	← **SP**

Errors

None

C

```
extern pascal TxtMaskRec GetErrGlobals()
```

$140C GetErrorDevice

Returns the type of driver installed as the error output device. For BASIC or Pascal device drivers, also returns the slot number; for RAM-based drivers, returns a pointer to the jump table.

Parameters

Stack before call

Stack	
previous contents	
wordspace	**Word**—Space for result
-- *longspace* --	**Long**—Space for result
	← **SP**

Stack after call

Stack	
previous contents	
deviceType	**Word**—Error output device type (0 = BASIC, 1 = Pascal, 2 = RAM-based)
-- *ptrOrSlot* --	**Long**—Slot number for BASIC or Pascal driver, or POINTER to jump table for RAM-based driver
	← **SP**

Errors

None

C

```
extern pascal DeviceRec GetErrorDevice()
```

$0C0C GetInGlobals

Returns the current values for the input device global parameters.

Parameters

Stack before call

Stack	
previous contents	
wordspace	**Word**—Space for result
wordspace	**Word**—Space for result
	← **SP**

Stack after call

Stack	
previous contents	
andMask	**Word**—Global AND mask for input device (low-order byte only)
orMask	**Word**—Global OR mask for input device (low-order byte only)
	← **SP**

Errors

None

C

```
extern pascal TxtMaskRec GetInGlobals()
```

$120C GetInputDevice

Returns the type of driver installed as the input device. For BASIC or Pascal device drivers, also returns the slot number; for RAM-based drivers, returns a pointer to the jump table.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *longspace* --	**Long**—Space for result
	← **SP**

Stack after call

previous contents	
deviceType	**Word**—Input device type (0 = BASIC, 1 = Pascal, 2 = RAM-based)
-- *ptrOrSlot* --	**Long**—Slot number for BASIC or Pascal driver, or POINTER to jump table for RAM-based driver
	← **SP**

Errors

None

C

```
extern pascal DeviceRec GetInputDevice()
```

$0D0C GetOutGlobals

Returns the current values for the output device global parameters.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
andMask	**Word**—Global AND mask for output device (low-order byte only)
orMask	**Word**—Global OR mask for output device (low-order byte only)
	← **SP**

Errors

None

C

```
extern pascal TxtMaskRec GetOutGlobals()
```

$130C GetOutputDevice

Returns the type of driver installed as the output device. For BASIC or Pascal device drivers, also returns the slot number; for RAM-based drivers, returns a pointer to the jump table.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *longspace* --	**Long**—Space for result
	← **SP**

Stack after call

previous contents	
deviceType	**Word**—Output device type (0 = BASIC, 1 = Pascal, 2 = RAM-based)
-- *ptrOrSlot* --	**Long**—Slot number for BASIC or Pascal driver, or POINTER to jump table for RAM-based driver
	← **SP**

Errors

None

C

```
extern pascal DeviceRec GetOutputDevice()
```

$150C InitTextDev

Initializes a specified text device.

Parameters

Stack before call

previous contents
deviceNum — **Word**—Device to initialize (0 = input, 1 = output, 2 = error output)
← **SP**

Stack after call

previous contents
← **SP**

Errors

$0C01 `badDevType` Illegal device type

C

```
extern pascal void InitTextDev(deviceNum)

Word     deviceNum;
```

$220C ReadChar

Reads a character obtained from the input text device, combines it with the input global masks, and returns the character on the stack. If *echoFlag* is set to a value of $0001, the character is also written to the output device.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
echoFlag	**Word**—Echo or don't echo character to output device (see Figure 23-1)
	← **SP**

Stack after call

previous contents	
char	**Word**—Character (in low-order byte)
	← **SP**

Errors

Pascal device errors — Any of the errors $0C02–$0C40 as listed in the section "Text Tool Set Summary" at the end of this chapter

C

```
extern pascal Word ReadChar(echoFlag)

Word     echoFlag;
```

The echo-flag word

The values for *echoFlag* are illustrated in Figure 23-1.

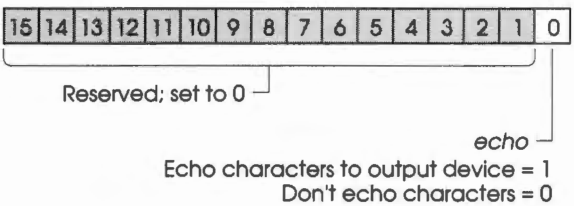


Figure 23-1
Character echo-flag word

$240C ReadLine

Reads a character string from the input text device, combines it with the input global masks, and writes the string to the memory location starting at *bufferPtr*. The character string is terminated when one of the following conditions is met:

- ☐ An End-of-line (EOL) character is received.
- ☐ The count of characters received is equal to the maximum line length specified by *maxCount*.

If *echoFlag* is set to a value of $0001, the characters are also written to the output device. The echoFlag is illustrated in Figure 23-1 in the section "ReadChar" in this chapter.

The count of characters received is returned on the stack.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *bufferPtr* --	**Long**—POINTER to start of buffer where characters will be written
maxCount	**Word**—Maximum line length
eolChar	**Word**—End-of-line character in low byte
echoFlag	**Word**—Echo or don't echo characters to output device (see Figure 23-1)
	← **SP**

Stack after call

previous contents	
charCount	**Word**—Count of characters received
	← **SP**

Errors

Pascal device errors	Any of the errors $0C02–$0C40 as listed in the section "Text Tool Set Summary" at the end of this chapter

C

```
extern pascal Word ReadLine(bufferPtr,maxCount,eolChar,echoFlag)
Pointer     bufferPtr;
Word      maxCount;
Word      eolChar;
Word      echoFlag;
```

$0B0C SetErrGlobals

Sets the global parameters for the error output device.

Parameters

Stack before call

previous contents
andMask — **Word**—Global AND mask for error output device (low-order byte only)
orMask — **Word**—Global OR mask for error output device (low-order byte only)
← **SP**

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void SetErrGlobals(andMask,orMask)

Word     andMask;

Word     orMask;
```

You can also use the following alternate form of the call:

```
extern pascal void SetErrGlobals(mask)

TxtMaskRec     mask;
```

$110C SetErrorDevice

Sets the error output device to a specified type and location. The routine returns an error if the *deviceType* specified is greater than 2.

Parameters

Stack before call

previous contents	
deviceType	**Word**—Error output device (0 = BASIC, 1 = Pascal, 2 = RAM based)
-- *ptrOrSlot* --	**Long**—Slot number for BASIC or Pascal driver, or POINTER to jump table for RAM-based driver
← **SP**	

Stack after call

Stack
previous contents
← **SP**

Errors

$0C01	`badDevType`	Illegal device type

C

```
extern pascal void SetErrorDevice(deviceType,ptrOrSlot)

Word      deviceType;

LongWord     ptrOrSlot;
```

You can also use the following alternate form of the call:

```
extern pascal void SetErrorDevice(deviceRec)

DeviceRec     deviceRec;
```

$090C SetInGlobals

Sets the global parameters for the input device.

Parameters

Stack before call

previous contents	
andMask	**Word**—Global AND mask for input device (low-order byte only)
orMask	**Word**—Global OR mask for input device (low-order byte only)
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetInGlobals(andMask,orMask)

Word      andMask;

Word      orMask;
```

You can also use the following alternate form of the call:

```
extern pascal void SetInGlobals(mask)

TxtMaskRec     mask;
```

$0F0C SetInputDevice

Sets the input device to a specified type and location. The routine returns an error if the *deviceType* is greater than 2.

Parameters

Stack before call

previous contents	
deviceType	**Word**—Input device type (0 = BASIC, 1 = Pascal, 2 = RAM based)
-- *ptrOrSlot* --	**Long**—Slot number for BASIC or Pascal driver, or POINTER to jump table for RAM-based driver
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$0C01	badDevType	Illegal device type

C

```
extern pascal void SetInputDevice(deviceType,ptrOrSlot)

Word      deviceType;

LongWord      ptrOrSlot;
```

You can also use the following alternate form of the call:

```
extern pascal void SetInputDevice(deviceRec)

DeviceRec     deviceRec;
```

(continued)

Assembly-language examples

BASIC example

```
PEA     0000          ; BASIC type
PEA     000           ; in slot 3
PEA     003
_SetInputDevice
```

RAM-based example

```
PEA     002           ; RAM-based type
PEA     label/256     ; RAM driver location
PEA     label
_SetInputDevice
```

$0A0C SetOutGlobals

Sets the global parameters for the output device.

Parameters

Stack before call

previous contents
andMask
orMask
← **SP**

andMask — **Word**—Global AND mask for output device (low-order byte only)

orMask — **Word**—Global OR mask for output device (low-order byte only)

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void SetOutGlobals(andMask,orMask)

Word      andMask;

Word      orMask;
```

You can also use the following alternate form of the call:

```
extern pascal void SetOutGlobals(mask)

TxtMaskRec     mask;
```

$100C SetOutputDevice

Sets the output device to a specified type and location. The routine returns an error if the *deviceType* specified is greater than 2.

Parameters

Stack before call

previous contents	
deviceType	**Word**—Output device type (0 = BASIC, 1 = Pascal, 2 = RAM based)
-- *ptrOrSlot* --	**Long**—Slot number for BASIC or Pascal driver, or POINTER to jump table for RAM-based driver
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

$0C01 `badDevType` Illegal device type

C

```
extern pascal void SetOutputDevice(deviceType,ptrOrSlot)

Word      deviceType;

LongWord      ptrOrSlot;
```

You can also use the following alternate form of the call:

```
extern pascal void SetOutputDevice(deviceRec)

DeviceRec      deviceRec;
```

$170C StatusTextDev

Executes a status call to a specified text device. The routine returns an error if the device is not ready.

Warning

BASIC devices do not support this routine.

Parameters

Stack before call

previous contents	
deviceNum	**Word**—Device for inquiry (0 = input, 1 = output, 2 = error output)
requestCode	**Word**—Request code (in low-order byte) to pass to *deviceNum*
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

Pascal device errors — Any of the errors \$0C02–\$0C40 as listed in the section "Text Tool Set Summary" at the end of this chapter

C

```
extern pascal void StatusTextDev(deviceNum,requestCode)

Word     deviceNum;

Word     requestCode;
```

$230C TextReadBlock

Reads a block of characters from the input text device, combines them with the input global masks, and writes the block to the memory location at *bufferPtr* + *offset*.

If *echoFlag* is set to a value of $0001, the characters are also written to the output device. The echoFlag word is illustrated in Figure 23-1 in the section "ReadChar" in this chapter.

Parameters

Stack before call

previous contents	
-- *bufferPtr* --	**Long**—POINTER to start of block
offset	**Word**—Offset to start of block to be read
blockSize	**Word**—Size of block to be read
echoFlag	**Word**—Echo or don't echo characters to output device (see Figure 23-1)
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

Pascal device errors	Any of the errors $0C02–$0C40 as listed in the section "Text Tool Set Summary" at the end of this chapter

C

```
extern pascal void TextReadBlock(bufferPtr,offset,blockSize,echoFlag)

Pointer     bufferPtr;

Word      offset;

Word      blockSize;

Word      echoFlag;
```

$1E0C TextWriteBlock

Combines a specified character string with the output global masks and then writes the string to the output text device. The string is specified by the *textPtr* + *offset* parameters, with the length specified by the *count* parameter.

Parameters

Stack before call

previous contents	
-- *textPtr* --	**Long**—POINTER to start of ASCII text
offset	**Word**—Offset within ASCII text
count	**Word**—String length
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

Pascal device errors — Any of the errors $0C02–$0C40 as listed in the section "Text Tool Set Summary" at the end of this chapter

C

```
extern pascal void TextWriteBlock(textPtr,offset,count)

Pointer    textPtr;

Word     offset;

Word     count;
```

$180C WriteChar

Combines a specified character with the output global AND mask and OR mask and writes the character to the output text device.

Parameters

Stack before call

previous contents
theChar — **Word**—Character to write (in low-order byte)
← **SP**

Stack after call

previous contents
← **SP**

Errors

Pascal device errors	Any of the errors $0C02–$0C40 as listed in the section "Text Tool Set Summary" at the end of this chapter

C

```
extern pascal void WriteChar(theChar)

Word     theChar;
```

$200C WriteCString

Combines a pointed-to C-type string (string terminates with $00) with the output global masks and then writes the string to the output text device.

Parameters

Stack before call

previous contents
-- *cStrPtr* --

Long—POINTER to C-type string to be written

← **SP**

Stack after call

previous contents

← **SP**

Errors

Pascal device errors	Any of the errors $0C02–$0C40 as listed in the section "Text Tool Set Summary" at the end of this chapter

C

```
extern pascal void WriteCString(cStrPtr)

Pointer     cStrPtr;
```

$1A0C WriteLine

Combines a pointed-to Pascal-type string (first byte of string specifies length) with the output global masks and then writes the string to the output text device. For BASIC and RAM-based drivers, the routine concatenates a carriage return to the string. For Pascal drivers, the routine concatenates a carriage return and a line feed to the string.

Parameters

Stack before call

previous contents
-- *strPtr* --
← **SP**

strPtr: **Long**—POINTER to Pascal-type string to be written

Stack after call

previous contents
← **SP**

Errors

Pascal device errors	Any of the errors $0C02–$0C40 as listed in the section "Text Tool Set Summary" at the end of this chapter

C

```
extern pascal void WriteLine(strPtr)

Pointer     strPtr;
```

$1C0C WriteString

Combines a pointed-to Pascal-type string (first byte of string specifies length) with the output global masks and then writes the string to the output text device.

Parameters

Stack before call

previous contents
-- *strPtr* --

strPtr: **Long**—POINTER to Pascal-type string to be written

← **SP**

Stack after call

previous contents

← **SP**

Errors

Pascal device errors — Any of the errors \$0C02–\$0C40 as listed in the section "Text Tool Set Summary" at the end of this chapter

C

```
extern pascal void WriteString(strPtr)

Pointer     strPtr;
```

Text Tool Set summary

This section briefly summarizes the constants and tool set errors contained in the Text Tool Set. There are no predefined data structures for the Text Tool Set.

Important

These definitions are provided in the appropriate interface file.

Table 23-4
Text Tool Set constants

Name	Value	Description
Echo flag values		
`noEcho`	$0000	Don't echo characters to output device
`echo`	$0001	Echo characters to output device
Device numbers		
`input`	$0000	Input device
`output`	$0001	Output device
`errorOutput`	$0002	Error output device
Device types		
`basicType`	$0000	Basic device type
`pascalType`	$0001	Pascal device type
`ramBased`	$0002	RAM-based device type

Table 23-5
Text Tool Set error codes

Code	Name	Description
$0C01	badDevType	Illegal device type
$0C02	badDevNum	Illegal device number
$0C03	badMode	Illegal operation
$0C04	unDefHW	Undefined hardware error
$0C05	lostDev	Lost device: device no longer on-line
$0C06	lostFile	File no longer in diskette directory
$0C07	badTitle	Illegal filename
$0C08	noRoom	Insufficient space on specified diskette
$0C09	noDevice	Specified volume not on-line
$0C0A	noFile	Specified file not in directory of specified volume
$0C0B	dupFile	Duplicate file: attempt to rewrite a file when a file of that name already exists
$0C0C	notClosed	Attempt to open file that is already open
$0C0D	notOpen	Attempt to access a closed file
$0C0E	badFormat	Error in reading real or integer number
$0C0F	ringBuffOFlo	Ring buffer overflow: characters arriving faster than the input buffer can accept them
$0C10	writeProtected	Specified diskette is write-protected
$0C40	devErr	Device error: device failed to complete a read or write correctly

Note: With the exception of `badDevType`, the errors should occur only for Pascal devices.

Chapter 24

Tool Locator

The **Tool Locator** is the tool set that allows your application to use tool sets without knowing whether the tool sets are in RAM or ROM.

In this chapter, we provide the information you need to use the existing tool sets. If you are writing your own tool set, however, you'll need to understand more about how the Tool Locator works. We have provided most of that information in Appendix A, "Writing Your Own Tool Set," as a special subject; please refer to that appendix for more information.

A preview of the Tool Locator routines

To introduce you to the capabilities of the Tool Locator, all Tool Locator routines are grouped by function and briefly described in Table 24-1. These routines are described in detail later in this chapter, where they are separated into housekeeping routines (discussed in routine number order) and the rest of the Tool Locator routines (discussed in alphabetical order).

Table 24-1
Tool Locator routines and their functions

Routine	Description
Housekeeping routines	
TLBootInit	Initializes the Tool Locator and all other ROM-based tool sets—must not be called by an application
TLStartUp	Starts up the Tool Locator for use by an application
TLShutDown	Shuts down the Tool Locator when an application quits
TLVersion	Returns the version number of the Tool Locator
TLReset	Resets the Tool Locator and all other ROM-based tool sets when the system is reset—must not be called by an application
TLStatus	Indicates whether the Tool Locator is active
Tool locating routines	
LoadTools	Ensures that specified system tool sets are available and have specified minimum version numbers
LoadOneTool	Ensures that a specified system tool set is available and has a specified minimum version number
UnloadOneTool	Unloads a specified tool set from memory
GetFuncPtr	Returns an entry in the function pointer table for a specified function in a specified tool set
GetTSPtr	Returns the pointer to the function pointer table of a specified tool set
SetTSPtr	Installs the pointer to a function pointer table in the appropriate tool pointer table
GetWAP	Gets the pointer to the work area for a specified tool set
SetWAP	Sets the pointer to the work area for a specified tool set
TLMountVolume	Displays on the Super Hi-Res display a simulated dialog box that your application can use to ask the user to mount a volume
TLTextMountVolume	Displays on the 40-column text screen a simulated dialog box that your application can use to ask the user to mount a volume
SaveTextState	Saves the state of the text screen and forces the hardware to display the text screen regardless of the display mode in use
RestoreTextState	Restores the state of the text screen from a specified handle and disposes of the handle
MessageCenter	Allows applications to communicate with each other

Using the Tool Locator

This section discusses how some of the Tool Locator routines fit into the general flow of an application and gives you an idea of which routines you'll need to use under normal circumstances. Each routine is described in detail later in this chapter.

The Tool Locator does not depend on the presence of any of the other tool sets; rather, all other tool sets depend on the Tool Locator.

The first Tool Locator call your application must make is TLStartUp. TLStartUp starts the Tool Locator mechanism that allows the other tool sets to be found. Conversely, when you quit your application, you must make the TLShutDown call just before the application quits.

Your application ensures that the tool sets it needs are available—and that they meet specified minimum version requirements—by making a LoadTools call. If, for example, you wanted to use the LineEdit Tool Set routine LETextBox2, which is available only in versions 2.0 and later of the tool set, you would specify 2.0 as the minimum version for the LineEdit Tool Set in the LoadTools call. The LoadTools routine loads all the tool sets that meet the minimum version requirements of those specified in the routine, thus making them available for the life of the application.

RAM-based tool sets are loaded only when the application requests they be loaded. Thus, if you want to load a tool set just before your application uses it, you can use the LoadOneTool routine (this routine also checks for the minimum tool set version). When you are finished with the tool set, you can unload it by using the UnloadOneTool routine.

The routines TLMountVolume and TLTextMountVolume can help if the boot disk containing the specified tool set (in the TOOLS subdirectory of the SYSTEM directory) is not available when the tool set is requested. TLMountVolume provides a mechanism your application can use to display a prompting message and OK and Cancel buttons on the Super Hi-Res display. TLTextMountVolume performs the same function (with a message only, no buttons) for the 40-column text display.

You can use the SaveTextState and RestoreTextState routines to save and restore the state of the text screen. You can use the MessageCenter call to communicate with other applications.

Normally, your application won't need to use the GetWap, SetWap, GetTSPtr, SetTSPtr, and GetFuncPtr calls; those calls are used only when you are writing your own tool set (see Appendix A, "Writing Your Own Tool Set," for more details).

$0101 TLBootInit

Initializes the Tool Locator and all other ROM-based tool sets; called only by the system startup firmware.

Warning
An application must never make this call.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C Call must not be made by an application.

$0201 TLStartUp

Starts up the Tool Locator for use by an application.

Important
Your application must make this call before it makes any other tool calls.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void TLStartUp()
```

$0301 TLShutDown

Shuts down the Tool Locator when an application quits.

Important
Your application must make this call just before it quits.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void TLShutDown()
```

$0401 TLVersion

Returns the version number of the Tool Locator.

Parameters

Stack before call

Stack	
previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

Stack	
previous contents	
versionInfo	**Word**—Version number of Tool Locator
	← **SP**

Errors None

C

```
extern pascal Word TLVersion()
```

$0501 TLReset

Resets the Tool Locator and all other ROM-based tool sets when the system is reset; called only by the system firmware.

Warning
An application must never make this call.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C Call must not be made by an application.

$0601 TLStatus

Indicates whether the Tool Locator is active.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

previous contents	
activeFlag	**Word**—BOOLEAN; TRUE (Tool Locator is always active)
	← **SP**

Errors None

C
```
extern pascal Boolean TLStatus()
```

$0B01 GetFuncPtr

Returns an entry in the function pointer table for a specified function in a specified tool set.

❖ *Note:* The word containing *funcNum* and *tsNum* is the same as the number the Tool Locator normally uses to locate the routine. For example, the appropriate number for GetFuncPtr is $0B01.

Parameters

Stack before call

previous contents		
longspace		**Long**—Space for result
userOrSystem		**Word**—$0000 = system tool set; $8000 = user tool set
funcNum	*tsNum*	**Byte**—Function number | **Byte**—Tool set number
		← **SP**

Stack after call

previous contents	
fptEntry	**Long**—Entry in function pointer table
	← **SP**

Errors

$0001	`toolNotFoundErr`	Specified tool set not found
$0002	`funcNotFoundErr`	Specified routine not found

C

```
extern pascal Pointer GetFuncPtr(userOrSystem,funcTSNum)
Word     userOrSystem;
Word     funcTSNum;
```

$0901 GetTSPtr

Returns the pointer to the function pointer table of a specified tool set.

❖ *Note:* This call is normally used only if you are writing your own tool set. See Appendix A, "Writing Your Own Tool Set," for more information.

Parameters

Stack before call

Stack	
previous contents	
-- *longspace* --	**Long**—Space for result
userOrSystem	**Word**—$0000 = system tool set; $8000 = user tool set
tsNum	**Word**—Tool set number of tool set whose pointer is to be returned
	← **SP**

Stack after call

Stack	
previous contents	
-- *fptPtr* --	**Long**—POINTER to function pointer table of tool set
	← **SP**

Errors

$0001 `toolNotFoundErr` Specified tool set not found

C

```
extern pascal Pointer GetTSPtr(userOrSystem,tsNum)
Word      userOrSystem;
Word      tsNum;
```

$0C01 GetWAP

Gets the pointer to the work area for a specified tool set.

❖ *Note:* This call is normally used only if you are writing your own tool set. See Appendix A, "Writing Your Own Tool Set," for more information.

Parameters

Stack before call

Stack	
previous contents	
— *longspace* —	**Long**—Space for result
userOrSystem	**Word**—$0000 = system tool set; $8000 = user tool set
tsNum	**Word**—Number of tool set
	← **SP**

Stack after call

Stack	
previous contents	
— *waptPtr* —	**Long**—POINTER to work area of tool set
	← **SP**

Errors

$0001 `toolNotFoundErr` Specified tool set not found

C

```
extern pascal Pointer GetWAP(userOrSystem,tsNum)

Word     userOrSystem;

Word     tsNum;
```

$0F01 LoadOneTool

Ensures that a specified system tool set is available (loading it from disk if necessary) and has at least a specified minimum version number. If the minimum version of the tool is not available in the TOOLS subdirectory of the SYSTEM directory in the boot volume, an error occurs.

❖ *Note:* See the section "LoadTools" in this chapter for a list of tool set numbers.

Parameters

Stack before call

previous contents	
toolNumber	**Word**—INTEGER; tool set number of tool set to load
minVersion	**Word**—Minimum version number of tool set needed
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

$0001	`toolNotFoundErr`	Specified tool set not found
$0010	`toolVersionErr`	Specified minimum version not found
System Loader errors		Returned unchanged
ProDOS errors		Returned unchanged

C

```
extern pascal void LoadOneTool(toolNumber,minVersion)

Word     toolNumber;

Word     minVersion;
```

$0E01 LoadTools

Ensures that specified system tool sets are available and checks that the tool sets have at least a specified **minimum version number.** The call needs, as input, a pointer to a tool table. The tool table lists the total number of tool sets, the number of each tool set needed, and the minimum acceptable version number of each tool set.

The structure of the tool table is illustrated in Figure 24-1. The numbers of the tool sets are given in Table 24-2.

Important
RAM-based tools are loaded from the TOOLS subdirectory of the SYSTEM directory. Each tool set is a load file of type $BA and is named after its decimal tool set number (Tool 23 is in a file named TOOL023, and so on).

Parameters

Stack before call

Stack	
previous contents	
-- *toolTablePtr* --	**Long**—POINTER to table of tool set numbers
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

$0001	`toolNotFoundErr`	Specified tool set not found
$0010	`toolVersionErr`	Specified minimum version not found
System Loader errors		Returned unchanged
ProDOS errors		Returned unchanged

C

```
extern pascal void LoadTools(toolTablePtr)

Pointer     toolTablePtr;
```

(continued)

Tool table

The form the tool table must take is shown in Figure 24-1.

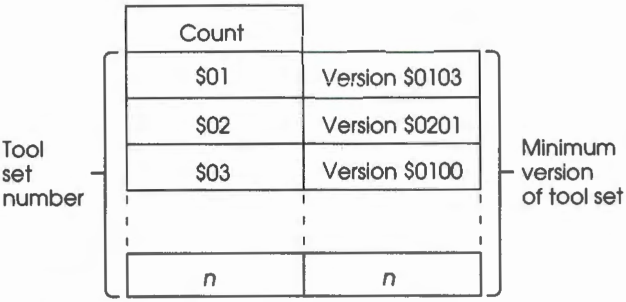


Figure 24-1
Tool table

Thus, in assembly language the table looks like this:

```
dc i'NumToolsRequired'

dc i'ToolNum1,MinVersion'

dc i'ToolNum2,MinVersion'

...

dc i'ToolNumN,MinVersion'
```

The format of the *minVersion* word is as follows:

High-order byte = Major version
Low-order byte = Minor version

Thus, a value of $0201 means a major version of 2 and a minor version of 1, and is commonly referred to as "Version 2.1."

Tool set numbers

The tool set numbers of all of the tool sets are listed in Table 24-2.

Table 24-2
Tool set numbers

Tool set number		Tool set name
$01	#01	Tool Locator
$02	#02	Memory Manager
$03	#03	Miscellaneous Tool Set
$04	#04	QuickDraw II
$05	#05	Desk Manager
$06	#06	Event Manager
$07	#07	Scheduler
$08	#08	Sound Tool Set
$09	#09	Apple Desktop Bus Tool Set
$0A	#10	SANE Tool Set
$0B	#11	Integer Math Tool Set
$0C	#12	Text Tool Set
$0D	#13	Reserved for internal use
$0E	#14	Window Manager
$0F	#15	Menu Manager
$10	#16	Control Manager
$11	#17	System Loader
$12	#18	QuickDraw II Auxiliary
$13	#19	Print Manager
$14	#20	LineEdit Tool Set
$15	#21	Dialog Manager
$16	#22	Scrap Manager
$17	#23	Standard File Operations Tool Set
$18	#24	Not available
$19	#25	Note Synthesizer
$1A	#26	Note Sequencer
$1B	#27	Font Manager
$1C	#28	List Manager

$1501 MessageCenter

Allows applications to communicate with each other.

Important

This routine is available only in Version 2.1 or later of the Tool Locator.

The message types are administered by Developer Technical Support at Apple Computer.

Parameters

Stack before call

previous contents	
action	**Word**—1 = add, 2 = get, 3 = delete (see Table 24-3)
type	**Word**—Message ID number
— *messageHandle* —	**Long**—HANDLE to message
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

$0111 `messNotFoundErr` Specified message not found

C

```
extern pascal void MessageCenter(action,type,messageHandle)
Word      action;
Word      type;
Handle     messageHandle;
```

More about messages

The action codes the routine will recognize are shown in Table 24-3.

Table 24-3
MessageCenter action codes

Type	Name	Description
1	`addMessage`	Adds the specified message to the message center data. The *type* is the ID of the message being added; any message already in the message center with this type is deleted. This message does not alter the *messageHandle*.
2	`getMessage`	Returns the specified message from the message center. If there is no message with the specified ID, an error occurs and the *messageHandle* is not altered. The *messageHandle* block can be of any size; the routine will resize the block to fit the data being returned.
3	`deleteMessage`	Deletes a specified message from the message center. If the message does not appear in the message center, an error occurs. The *messageHandle* is not used for this call, and any value may be passed.

The actual data in the message must begin with six bytes of reserved space, as follows:

```
MessNext    LONG     Handle of next message
MessType    WORD     Type of this message
MessData    block    Message-specific data
```

The only message defined at the time of publication is type 1. That message type is used to pass file information from one application to another. For example, a Finder-type application needs to tell other applications what documents it should open or print. In this case, the message data is set up as follows:

```
MessNext    LONG
MessType    WORD     1; file info type
MessData    block    0 = open the following files
                     1 = print the following files
            str 'name1'
            str 'full pathname 2'
            . . .
            BYTE     0
```

$1401 RestoreTextState

Restores the state of the text screen from a specified handle and disposes of the handle. The state of the text screen can be saved with a SaveTextState call; see the section "SaveTextState" in this chapter.

Parameters

Stack before call

previous contents	
-- *stateHandle* --	**Long**—HANDLE to state record to restore
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void RestoreTextState(stateHandle)

Handle      stateHandle;
```

$1301 SaveTextState

Saves the state of the text screen and forces the hardware to display the text screen regardless of the display mode in use. The routine does not initialize the Text Tool Set.

Important

This routine is available only in Version 2.1 or later of the Tool Locator.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
	← **SP**

Stack after call

previous contents	
-- *stateHandle* --	**Long**—HANDLE to state record (see Table 24-4)
	← **SP**

Errors

None

C

```
extern pascal Handle SaveTextState()
```

(continued)

The SaveTextState record

The structure of the record in which the SaveTextState information is saved is private. At the time of publication, the various entities saved included those shown in Table 24-4.

Table 24-4
State record

Entity	Description
StateReg	Language card information
80Col	40 or 80 columns
NewVideo	New video register
Text	Text state soft switch
Mix	Split mode soft switch
Page2	Page 1 or page 2
HiRes	HiRes or LoRes
AltCharSet	Character set
80Vid	State of 80 store
Cursor	Internal cursor variable
c3ROM	State of the slot 3 firmware
OutputDevice	Text tools output device
OutGlobals	Text tools output globals
InputDevice	Text tools input device
InGlobals	Text tools input globals
Screen Memory in bank $0	
Screen Memory in bank $1	
Screen Memory in bank $E0	
Screen Memory in bank $E1	
Screen holes for Slot 3	

$0A01 SetTSPtr

Installs the pointer to a function pointer table in the appropriate tool pointer table (TPT).

❖ *Note:* This call is normally used only if you are writing your own tool set. See Appendix A, "Writing Your Own Tool Set," for more information.

If the TPT is not yet in RAM, the routine copies it to RAM. (Memory for the TPT is obtained from the Memory Manager.) If there is not enough room in the TPT for the new entry, the TPT is moved to a bigger chunk of memory. Likewise, the Work Area Pointer Table (WAPT) is expanded, if necessary (memory for the expansions is obtained from the Memory Manager).

If the new pointer table has any 0 entries, old entries are moved from the old pointer table to the new pointer table. This feature makes it possible, for example, to patch just a portion of a tool set rather than replacing the tool set entirely.

Parameters

Stack before call

previous contents	
userOrSystem	**Word**—$0000 = system tool set, $8000 = user tool set
tsNum	**Word**—Tool set number of tool set
fptPtr	**Long**—POINTER to function pointer table for tool set
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

$0001	`toolNotFoundErr`	Specified tool set not found

C

```
extern pascal void SetTSPtr(userOrSystem,tsNum,fptPtr)

Word     userOrSystem;

Word     tsNum;

Pointer     fptPtr;
```

$0D01 SetWAP

Sets the pointer to the work area for a specified tool set.

❖ *Note:* This call is normally used only if you are writing your own tool set. See Appendix A, "Writing Your Own Tool Set," for more information.

Parameters

Stack before call

previous contents	
userOrSystem	**Word**—$0000 = system tool set; $8000 = user tool set
tsNum	**Word**—Tool set number of tool set
-- *waptPtr* --	**Long**—POINTER to work area of tool set
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

$0001 `toolNotFoundErr` Specified tool set not found

C

```
extern pascal void SetWAP(userOrSystem,tsNum,waptPtr)

Word      userOrSystem;

Word      tsNum;

Pointer      waptPtr;
```

$1101 TLMountVolume

Displays on the Super Hi-Res display a simulated dialog box that your application can use to ask the user to mount a volume. The routine also displays two buttons.

❖ *Note:* The box is not really a dialog box because it is not under the control of the Dialog Manager (a RAM-based tool set that might not be active).

The text displayed must be supplied by the application (and can therefore be easily translated into other languages). The contents of the screen under the box are saved before the box is drawn and are restored after the user responds. The button may be chosen by the user either by clicking on it with the mouse button or by pressing the Return key for button 1 or the Esc (escape) key for button 2.

Important

Your application must make sure that the text passed in the strings fits in the area provided.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
whereX	**Word**—INTEGER; upper left X coordinate for box
whereY	**Word**—INTEGER; upper left Y coordinate for box
-- *line1Ptr* --	**Long**—POINTER to Pascal-type string to appear at top of box
-- *line2Ptr* --	**Long**—POINTER to Pascal-type string to appear just below line 2
-- *but1Ptr* --	**Long**—POINTER to Pascal-type string to appear inside button 1
-- *but2Ptr* --	**Long**—POINTER to Pascal-type string to appear inside button 2
	← **SP**

Stack after call

previous contents
whichButton — **Word**—Button number chosen; Return = 1, Esc = 2
← **SP**

Errors None

C

```
extern pascal Word TLMountVolume(whereX,whereY,line1Ptr,line2Ptr,
but1Ptr,but2Ptr)

Integer     whereX;
Integer     whereY;
Pointer     line1Ptr;
Pointer     line2Ptr;
Pointer     but1Ptr;
Pointer     but2Ptr;
```

You can also use the following alternate form of the call:

```
extern pascal Word TLMountVolume(where,line1Ptr,line2Ptr,but1Ptr,but2Ptr)

Point       where;
Pointer     line1Ptr;
Pointer     line2Ptr;
Pointer     but1Ptr;
Pointer     but2Ptr;
```

$1201 TLTextMountVolume

Displays on the 40-column text screen a simulated dialog box that your application can use to ask the user to mount a volume. The routine also displays two buttons.

❖ *Note:* The box is not really a dialog box because it is not under the control of the Dialog Manager (a RAM-based tool set that might not be active).

The text displayed must be supplied by the application (and can therefore be easily translated into other languages). The contents of the screen under the box are saved before the box is drawn and restored after the user responds. The button is chosen by the user pressing the Return key for button 1 or the Esc (escape) key for button 2.

Important

Your application must make sure that the text passed in the strings fits in the area provided.

Parameters

Stack before call

Stack	
previous contents	
wordspace	**Word**—Space for result
line1Ptr	**Long**—POINTER to Pascal-type string to appear at top of box
line2Ptr	**Long**—POINTER to Pascal-type string to appear just below line 2
button1Ptr	**Long**—POINTER to Pascal-type string to appear inside button 1
button2Ptr	**Long**—POINTER to Pascal-type string to appear inside button 2
	← **SP**

Stack after call

Stack	
previous contents	
whichButton	**Word**—Button number chosen; Return = 1, Esc = 2
	← **SP**

Errors None

C

```
extern pascal Word TLTextMountVolume(line1Ptr,line2Ptr,but1Ptr,but2Ptr)
Pointer     line1Ptr;
Pointer     line2Ptr;
Pointer     but1Ptr;
Pointer     but2Ptr;
```

$1001 UnloadOneTool

Unloads a specified tool set from memory. See Table 24-2 in the section "LoadTools" in this chapter for the tool set numbers.

The tool set is left in a restartable state; that is, the purge level of all of the tool set's memory blocks is set to 3 so that the tool set may be restarted quickly. See Chapter 12, "Memory Manager," in Volume 1 for more information on purge levels.

Parameters

Stack before call

Stack	
previous contents	
toolNumber	**Word**—INTEGER; tool set number of tool set to unload
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

$0001 `toolNotFoundErr` Specified tool set not found

C

```
extern pascal void UnloadOneTool(toolNumber)

Word     toolNumber;
```

Tool Locator summary

This section briefly summarizes the constants and tool set error codes contained in the Tool Locator. There are no predefined data structures for the Tool Locator.

Important

These definitions are provided in the appropriate interface file.

Table 24-5
Tool Locator constants

Name	Value	Description
Message center action codes		
`addMessage`	$0001	Add message
`getMessage`	$0002	Get message
`deleteMessage`	$0003	Delete message
Message center type values		
`fileInfoType`	$0001	
TLMountVolume button values		
`mvReturn`	$0001	Equivalent of dialog OK button
`mvEscape`	$0002	Equivalent of dialog Cancel button
Tool set specification		
`sysTool`	$0000	System tool set
`userTool`	$8000	User tool set

Table 24-6
Tool Locator error codes

Code	Name	Description
$0001	`toolNotFoundErr`	Specified tool set not found
$0002	`funcNotFoundErr`	Specified routine not found
$0110	`toolVersionErr`	Specified minimum version not found
$0111	`messNotFoundErr`	Specified message not found

Chapter 25

Window Manager

The **Window Manager** is a tool set for dealing with windows on the Apple IIGS 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 in which information, such as a document or a picture, is presented. Windows can be any size or shape, and there can be one or many of them, depending on the application.

Windows allow the application to control more information than the screen can display at one time. The name *window* is used because the user sees through the window into a larger area, as illustrated in Figure 25-1.

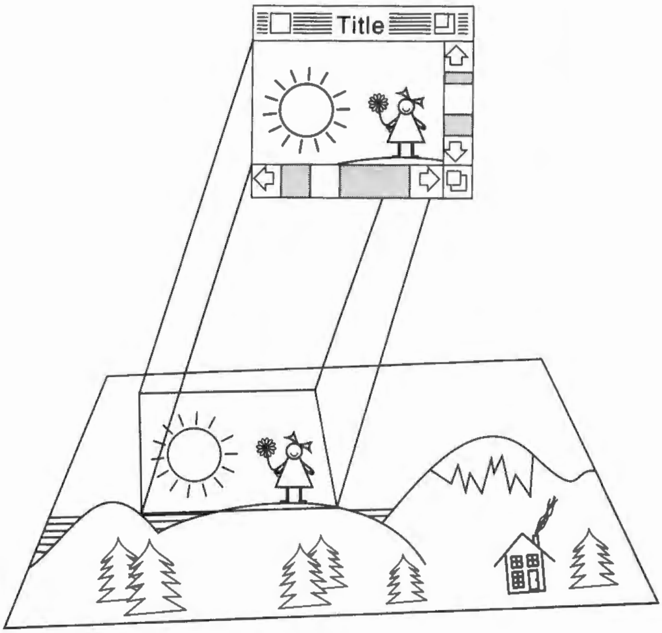


Figure 25-1
Window

A preview of the Window Manager routines

To introduce you to the capabilities of the Window Manager, all Window Manager routines are grouped by function and briefly described in Table 25-1. These routines are described in detail later in this chapter, where they are separated into housekeeping routines (discussed in routine number order) and the rest of the Window Manager routines (discussed in alphabetical order).

Table 25-1
Window Manager routines and their functions

Routine	Description
Housekeeping routines	
WindBootInit	Initializes the Window Manager; called only by the the Tool Locator—must not be called by an application
WindStartUp	Starts up the Window Manager for use by an application
WindShutDown	Shuts down the Window Manager when an application quits
WindVersion	Returns the version number of the Window Manager
WindReset	Resets the Window Manager; called only when the system is reset—must not be called by an application
WindStatus	Indicates whether the Window Manager is active
Initialization and termination routines	
Desktop	Controls the addition of regions to and subtraction of regions from the desktop and controls the current desktop pattern
NewWindow	Creates a specified window as specified by its parameters, adds it to the window list, and returns a pointer to the new window's GrafPort
CloseWindow	Removes a specified window from the screen, disposes of all controls associated with that window, and deletes the window from the window list
WindNewRes	Closes the Window Manager's GrafPort and opens a new GrafPort in the other Super Hi-Res resolution
Window record and global access routines	
GetWMgrPort	Returns a pointer to the Window Manager's port
SetWindowIcons	Sets the icon font for the Window Manager
SetWRefCon	Sets a value that is inside a specified window record and is reserved for the application's use
GetWRefCon	Returns a value from a a specified window's record that was passed to either NewWindow or SetWRefCon by the application
SetWTitle	Changes the title of a specified window to a specified title and redraws the window
GetWTitle	Returns the pointer to a specified window's title
SetFrameColor	Sets the color of a specified window's frame
GetFrameColor	Returns the color of a specified window's frame
FrontWindow	Returns a pointer to the first visible window in the window list (that is, the active window)

Table 25-1 (continued)
Window Manager routines and their functions

Routine	Description
Window record and global access routines	
GetFirstWindow	Returns a pointer to the first window in the window list (the window may not be the active window)
GetNextWindow	Returns a pointer to the next window in the window list after a specified window; returns NIL if the specified window is the last window in the window list
GetWKind	Indicates whether a specified window is a system window or an application window
SetWFrame	Sets the bit flag that describes a specified window's frame type
GetWFrame	Returns the bit flag that describes a specified window's frame type
GetStructRgn	Returns a handle to a specified window's structure region
GetContentRgn	Returns a handle to a specified window's content region
GetUpdateRgn	Returns a handle to a specified window's update region
GetDefProc	Returns a pointer to the routine that is called to draw, hit test, and otherwise define a window's frame and behavior
SetDefProc	Sets the pointer to the routine that is called to draw, hit test, and otherwise define a window's frame and behavior
GetWControls	Returns the handle to the first control in the window's control list
SetZoomRect	Sets the rectangle to be used as the content's zoomed or unzoomed size for a specified window
GetZoomRect	Returns a pointer to the rectangle to be used as the content's zoomed or unzoomed size for a specified window
GetSysWFlag	Indicates whether a specified window is a system or an application window
SetSysWindow	Marks a specified window as a system window
GetContentOrigin	Returns the values used by TaskMaster to set the origin of the window's GrafPort when handling an update event
SetContentOrigin	Sets the origin of the window's GrafPort when handling an update event
SetContentOrigin2	Sets the origin of the window's GrafPort when handling an update event and allows the application to scroll or not scroll the window's content region
SetOriginMask	Specifies the mask used to put the horizontal origin on a grid
StartDrawing	Makes a specified window the current port and sets its origin
GetDataSize	Returns the height and width of the data area of a specified window
SetDataSize	Sets the height and width of the data area of a specifed window
GetMaxGrow	Returns the maximum values to which a specified window's content region can grow
SetMaxGrow	Sets the maximum values to which a specified window's content region can grow
GetScroll	Returns the number of pixels by which TaskMaster will scroll the content region when the user selects the arrows on window frame scroll bars
SetScroll	Sets the number of pixels by which TaskMaster will scroll the content region when the user selects the arrows on window frame scroll bars

(continued)

Table 25-1 (continued)
Window Manager routines and their functions

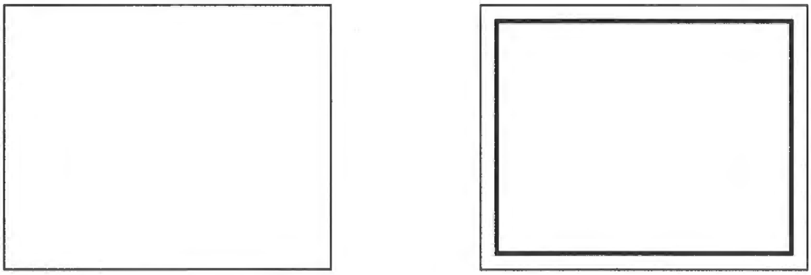
Routine	Description
Window record and global access routines	
GetPage	Returns the number of pixels by which TaskMaster will scroll the content region when the user selects the page regions on window frame scroll bars
SetPage	Sets the number of pixels by which TaskMaster will scroll the content region when the user selects the page regions on window frame scroll bars
GetContentDraw	Returns the pointer to the routine that draws the content region of a specified window
SetContentDraw	Sets the pointer to the routine to draw the content region of a specified window
Information bar routines	
GetInfoDraw	Returns the pointer to the routine that draws the information bar for a specified window
SetInfoDraw	Sets the pointer to the routine that draws the information bar for a specified window
GetInfoRefCon	Returns the value associated with the draw information bar routine for a specified window
SetInfoRefCon	Sets the value associated with the draw information bar routine for a specified window
GetRectInfo	Sets the information rectangle to the coordinates of the information bar rectangle
StartInfoDrawing	Allows an application to draw or hit test outside of its information bar definition procedure
EndInfoDrawing	Puts the Window Manager back into a global coordinate system
Window shuffling routines	
SelectWindow	Makes a specified window the active window
HideWindow	Makes a specified window invisible
ShowWindow	Makes a specified window visible if it was invisible and then draws the window
ShowHide	Shows or hides a window
BringToFront	Brings a specified window to the front of all other windows and redraws the windows as necessary but does not do any highlighting or unhighlighting
SendBehind	Changes the position of a specified window, redrawing any exposed windows
Window drawing routines	
HiliteWindow	Highlights or unhighlights a specified window
RefreshDesktop	Redraws the entire desktop and all the windows

Table 25-1 (continued)
Window Manager routines and their functions

Routine	Description
User interaction routines	
FindWindow	Indicates which part of which window, if any, the cursor was in when the user pressed the mouse button
DragWindow	Pulls around a dotted outline of a specified window, following the movements of the mouse until the mouse button is released
GrowWindow	Pulls around a grow image of a specified window, following the movements of the mouse until the mouse button is released
TrackGoAway	Tracks the mouse until the mouse button is released, highlighting the go-away region as long as the mouse location remains inside it and unhighlighting it when the mouse moves outside it
TrackZoom	Tracks the mouse until the mouse button is released, highlighting the zoom region as long as the mouse location remains inside it and unhighlighting it when the mouse moves outside it
TaskMaster	Calls GetNextEvent and looks in the event part of the task record to see if it can handle the event
Window sizing and positioning routines	
MoveWindow	Moves a specified window to another part of the screen without affecting its size
SizeWindow	Enlarges or shrinks the port rectangle of a specified window's GrafPort to a specified width and height
ZoomWindow	Switches the size and position of a specified window between its current size and position and its maximum size
WindDragRect	Pulls a dotted outline of a specified rectangle around the screen, following the movements of the mouse until the mouse button is released
Update region routines	
InvalRect	Accumulates a specified rectangle into the update region of the window whose GrafPort is the current port
InvalRgn	Accumulates a specified region into the update region of the window whose GrafPort is the current port
ValidRect	Removes a specified rectangle from the update region of the window whose GrafPort is the current port and tells the Window Manager to cancel any updates accumulated for that rectangle
ValidRgn	Removes a specified region from the update region of the window whose GrafPort is the current port and tells the Window Manager to cancel any updates accumulated for that region
BeginUpdate	Replaces the visible region of the window's GrafPort with the intersection of the visible region and the update region and then sets the window's update region to an empty region
EndUpdate	Restores the normal visible region of a specified window's GrafPort that was changed by a BeginUpdate call
Miscellaneous routines	
PinRect	Pins a specified point inside a specified rectangle
CheckUpdate	Looks from top to bottom in the window list for a visible window that needs updating (that is, for a window whose update region is not empty)

Window frames and controls

There are two kinds of predefined window frames, document and alert. The **alert window** is used by the Dialog Manager and is explained in Chapter 6, "Dialog Manager," in Volume 1. The document window, which is used by the Window Manager, is explained in this section. The two types of window frames are illustrated in Figure 25-2.



Document window frame

Alert window frame

Figure 25-2
Window frames

A **document window** may have any or all of the **standard window controls,** as listed below. The only restriction is that if there is a close or zoom box there must also be a title bar, and common sense would dictate that there only be a zoom box if there is a size box, although this is not a requirement. The standard controls include the following:

- **Title bar,** a rectangle at the top of the window that contains the window's title, may hold the close and zoom boxes, and can be a drag region for moving the window
- **Close box,** a small region in the title bar that the user can select to remove the window from the screen
- **Zoom box,** a small region in the title bar that the user can select to make the window its maximum size and to return it to its previous size and position
- **Right scroll bar,** which the user selects to scroll vertically through the data in the window
- **Bottom scroll bar,** which the user selects to scroll horizontally through the data in the window
- **Size box,** a small region in the lower right corner of the window; the user can drag the size box to change the size of the window
- **Information bar,** a place in which an application can display some information that won't be affected by the scroll bars

These standard controls, which can be used only for document windows and may not be added to alert windows, are illustrated in Figure 25-3.

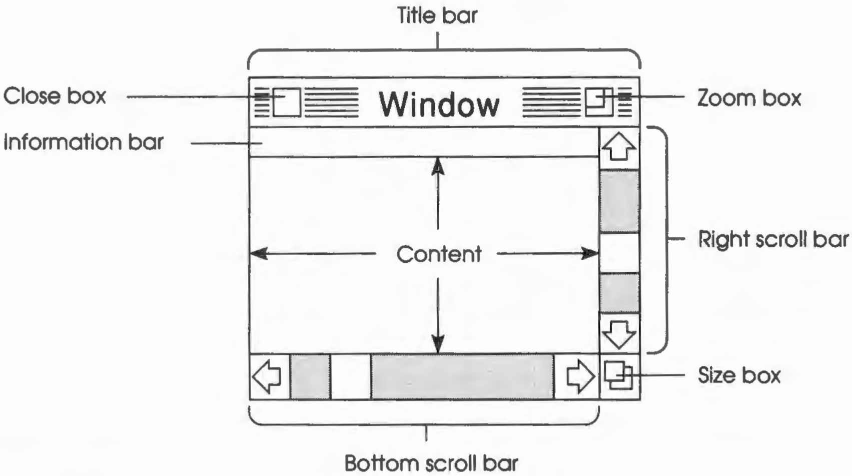


Figure 25-3
Standard window controls

Some possible document window combinations are illustrated in Figure 25-4.

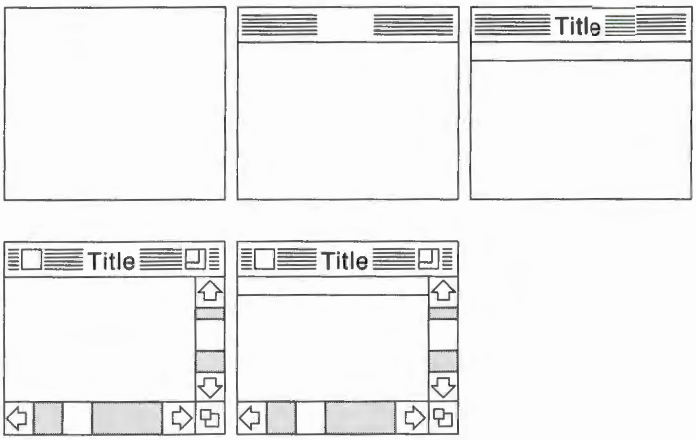


Figure 25-4
Sample document windows

You can either use the standard window types or create your own window types (see the section "Defining Your Own Windows" in this chapter). Some windows—such as the window the Dialog Manager creates to display an alert—may be created indirectly for you when you use other parts of the Toolbox. Windows created either directly or indirectly by an application are collectively called **application windows.** Another class of windows, called **system windows,** consists of 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 **planes** (front-to-back order), or change their sizes—all without concern for how the various windows overlap. The Window Manager keeps track of any newly exposed areas and provides a convenient mechanism with which you can ensure that they are properly redrawn.

You can also easily set up your application so that mouse actions cause the following standard responses inside a document window (or similar responses inside other windows):

- Clicking anywhere in an inactive window makes it the active window by bringing 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 that a representation of the window (such as an icon) is left on the desktop.
- Dragging anywhere inside the title bar of a window (except in a close or zoom box) pulls an outline of the window across the screen; 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 Apple 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 a **content region**—the area in which your application draws—and a **frame region**—the outline of the entire window plus any standard window controls. Together, the content and frame regions make up the **structure region.**

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

A window may also have any of the following regions within the window frame:

- **Go-away region,** a close box in the active window. Clicking in this region closes the window.
- **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 (unless it was already the active window or unless the Apple key was held down).
- **Grow region,** the size 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 (unless it was already the active window or unless the Apple key was held down).
- **Zoom region,** the zoom box in the active window. Clicking in this region toggles from the current position and size to a maximum size and back again.

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

Data and content areas and scroll bars

Windows act like a microfiche machine. What is seen in the window's content region is like what is seen on the viewer. Similarly, 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 the devices used for scrolling the data area through the content region and showing the relationship between the data area and content region. Because scroll bars are handled by the Control Manager, the Control Manager must be loaded and started up before scroll bars can be used in windows. The following paragraphs explain how standard window scroll bars act in relationship to windows.

The scroll bar is like a reduced cross section of the work area. The scroll thumb has the same ratio to the page region as the content region has to the data area, as illustrated in Figure 25-5.

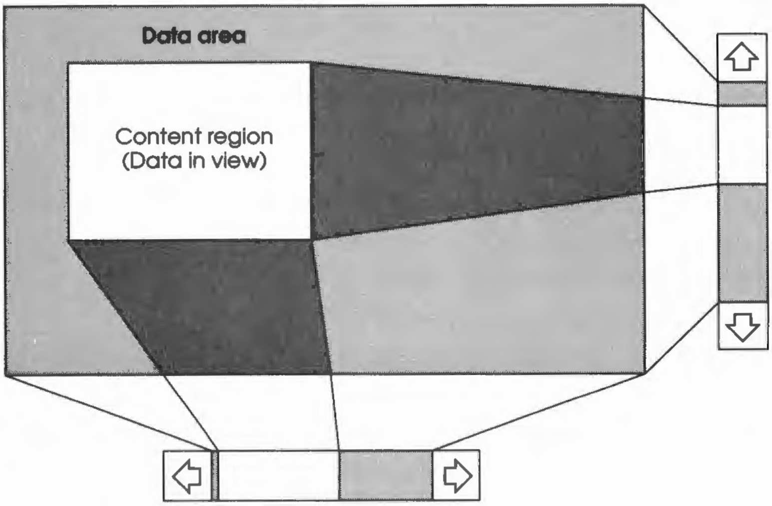


Figure 25-5
Proportional scroll bars

The size and zoom boxes are used to increase or decrease the amount of the data area displayed at one time. When the window is moved, the data area is moved with it so the view in the content region remains the same.

Using the Window Manager

This section discusses how the Window Manager routines fit into the general flow of an application and gives you an idea of which routines you'll need to use under normal circumstances. Each routine is described in detail later in this chapter.

The Window Manager depends on the presence of the tool sets shown in Table 25-2 and requires that at least the indicated version of each tool set be present.

Table 25-2
Window Manager—other tool sets required

Tool set number		Tool set name	Minimum version needed
$01	#01	Tool Locator	1.2
$02	#02	Memory Manager	1.2
$03	#03	Miscellaneous Tool Set	1.2
$04	#04	QuickDraw II	1.2
$06	#06	Event Manager	1.0

The first Window Manager call that your application must make is WindStartUp. Conversely, when you quit your application, you must make the WindShutDown call.

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

There are two ways to handle user input in relation to windows. You can poll the user via TaskMaster, which will handle most events that deal with standard user interfaces (see the section "Using TaskMaster" in this chapter).

If you are not using TaskMaster, you must poll for events by calling GetNextEvent in the Event Manager. Whenever your application receives an update event, the application should respond as follows:

1. Call the BeginUpdate routine. This routine temporarily replaces the visible region of the window's GrafPort with the intersection of the visible region and the update region. It then clears the update region for that window.
2. Draw the window contents.
3. Call the EndUpdate routine to restore the actual visible region.

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.
- Remove the highlighting or blinking cursor from text being edited when the window becomes inactive and restore it when the window becomes active.
- Enable or disable a menu or certain menu items appropriately to match what the user can do when windows become active or inactive.

If you are not using TaskMaster and a mouse-down event occurs, the application should 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:** Mouse-down somewhere outside the desktop. If you have not subtracted any area from the desktop, there is a good chance the button was pressed in the system menu bar. Call the Menu Manager routine MenuSelect.
- **wInDrag:** Mouse-down in a window's drag region; it may or may not be the active window. Call DragWindow.
- **wInContent:** Mouse-down in window's content region. Call SelectWindow if the window is not the active window. Otherwise, handle the event according to your application.
- **wInGoAway:** Mouse-down in active window's go-away region. Call TrackGoAway. If the routine returns TRUE, you may want to give the user the opportunity to save the window; then call CloseWindow or HideWindow.
- **wInZoom:** Mouse-down in active window's zoom region. Call TrackZoom. If TrackZoom returns TRUE, call ZoomWindow.
- **wInGrow:** Mouse-down in active window's grow region. Call GrowWindow.

Using TaskMaster

TaskMaster is a procedure that can handle many standard functions. When TaskMaster is called instead of GetNextEvent; the sequence of events is as follows:

1. TaskMaster calls GetNextEvent.
2. If there isn't an event ready, TaskMaster returns 0.

   If an event is ready, TaskMaster looks at it and tries to handle it.
3. If Taskmaster can't handle the event that is ready, it returns the event code to the application. The application can handle the event as if it had come from GetNextEvent.

   If TaskMaster can handle the event, it calls standard routines to try to complete the task. For example, if the user presses the mouse button in an active window's zoom region, TaskMaster detects it, calls TrackZoom, calls ZoomWindow (if the user actually selects the zoom region), and returns no event.

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 detects it but won't be able to go any further, so it returns `wInContent`, which tells the application the mouse button is down in the active window's content region.

We provide TaskMaster for two reasons. First, it should help you get an application running as quickly as possible and still allow you to take advantage of the standard user interface. TaskMaster should be usable by even the most advanced applications, although some alternate algorithms may have to be used to get the desired results.

Second, TaskMaster provides upward compatiblity in the years to come. If an application is using TaskMaster, a modification to TaskMaster to take advantage of some new feature will not adversely affect the application; in fact, your application may be using the new feature without any modification on your part!

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 passes the provided pointer so that the event record part of TaskRec is set by GetNextEvent. The structure of the task record is shown in Figure 25-6.

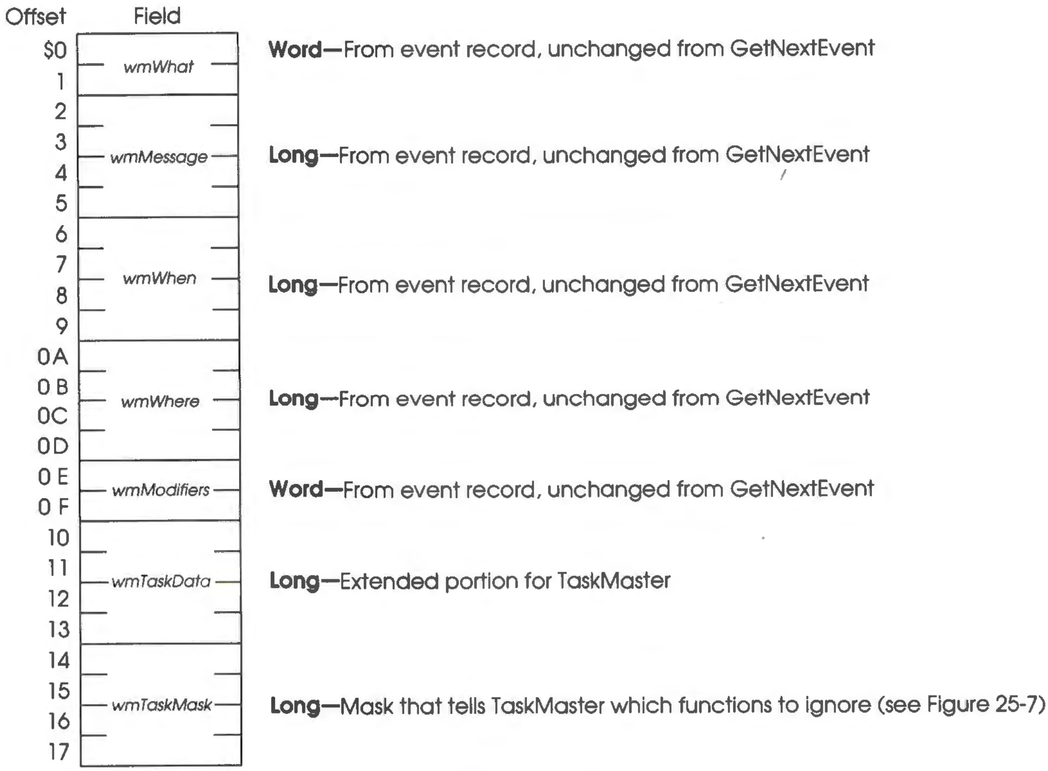


Figure 25-6
TaskMaster's TaskRec (task record)

The *wmtaskMask* is used by your application to tell TaskMaster about functions you would like it to ignore. The *wmtaskMask* is defined as shown in Figure 25-7.

Reserved at the time of publication; set to 0 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

tmInfo
Don't activate inactive window on click in info bar = 1
Activate inactive window on click in info bar = 0

tmInactive
Return `wInactMenu` when user selects inactive menu item = 1
Never return `wInactMenu` = 0

tmCRedraw
Redraw controls upon activate event = 1
Don't redraw controls upon activate event = 0

tmSpecial
Handle special menu items = 1
Don't handle special menu items = 0

tmScroll
Enable scrolling and activate inactive window on click in scroll bar = 1
Disable scrolling and don't activate window on click in scroll bar = 0

tmGrow
GrowWindow called = 1
GrowWindow not called = 0

tmZoom
TrackZoom called = 1
TrackZoom not called = 0

tmClose
TrackGoAway called = 1
TrackGoAway not called = 0

tmContent
Activate inactive window on click in content region = 1
Don't activate inactive window on click in content region = 0

tmDragW
DragWindow called = 1
DragWindow not called = 0

tmSysClick
SystemClick called = 1
SystemClick not called = 0

tmOpenDA
OpenNDA called = 1
OpenNDA not called = 0

tmMenuSel
MenuSelect called = 1
MenuSelect not called = 0

tmFindW
FindWindow called = 1
FindWindow not called = 0

tmUpdate
Update events handled = 1
Update events not handled = 0

tmMenuKey
MenuKey called = 1
MenuKey not called = 0

Figure 25-7
The *wmtaskMask* bit flag

Important

At the time of publication, bits 31–16 must be set to 0. In fact, TaskMaster will return an error if they are not. Because these bits will mask off as yet unknown features, applications will continue to run even when the new features are added.

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 to get the handle to the current font, call SetWindowIcons. The format of the font is shown in Table 25-3.

Table 25-3
Window Manager icon font

Character	Icon
0	Close box
1	Highlighted close and zoom boxes (same character for both)
2	Zoom box

Window record

The Window Manager keeps all the information it requires for its operations on a particular window in a **window record.** The record contains the information the Window Manager needs to manage windows. The complete window record is accessed directly only by the Window Manager. Your application can directly access only the part of the window record illustrated in Figure 25-8.

Not allowing direct access to the entire window record has advantages and disadvantages. Access to window information is slower if calls have to be made to the Window Manager. However, the delay could only be measured in milliseconds and can't be seen on the screen. On the plus side, future Window Managers won't be tied to an older, possibly inadequate, record structure. The chances of improving the current Window Manager without affecting existing applications and of maintaining compatibility across future hardware are greatly improved by allowing the window record to change.

Many Window Manager calls need as input a window pointer that is returned from NewWindow. That pointer points to the window's GrafPort.

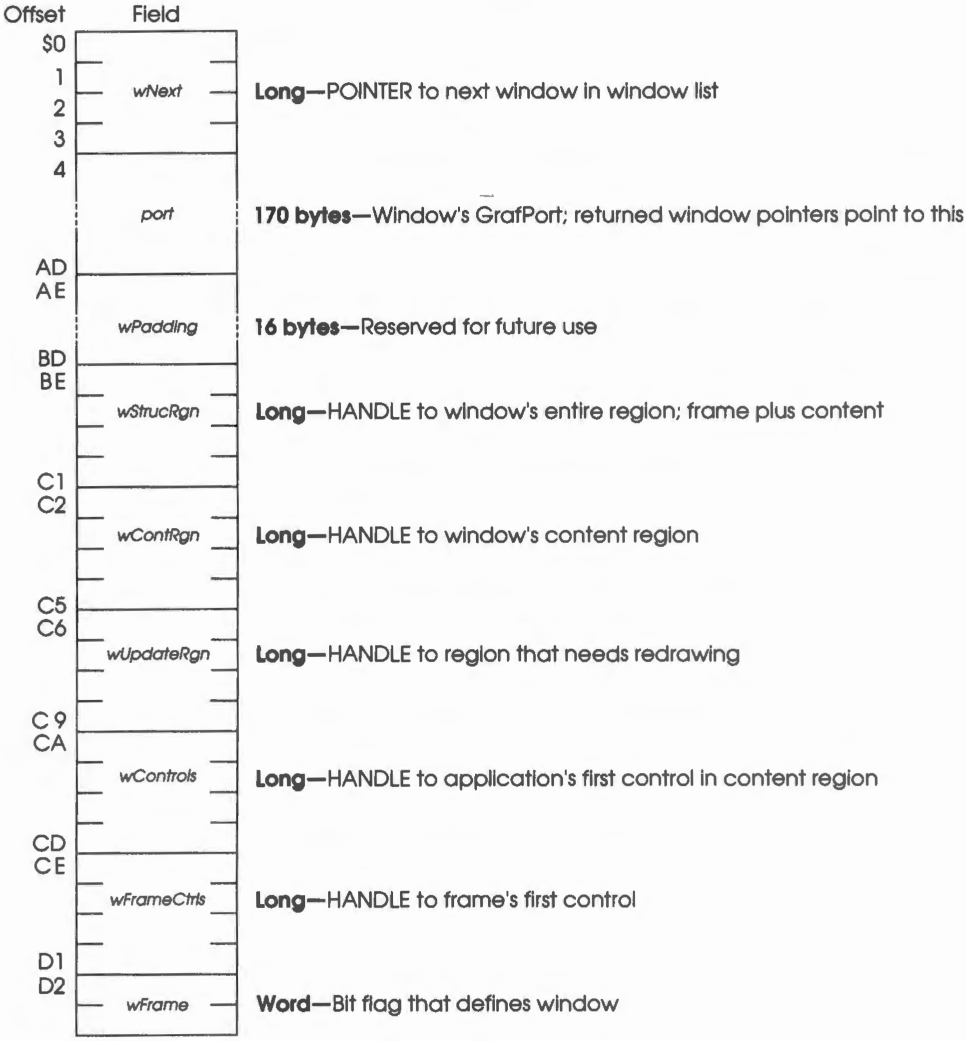


Figure 25-8
Window record

The settings for the *wFrame* parameter are described in the section "NewWindow" in this chapter.

Windows and GrafPorts

It's easy for your application to use windows; to the application, a window is a GrafPort the application can draw into with QuickDraw II routines. When you create a window, you specify a RECT data structure that becomes the port rectangle 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 II. These characteristics apply whenever the application draws in the window, and they can easily be changed with QuickDraw II routines.

There is, however, more to a window than just the GrafPort in which the application draws. The other part of a window is called the **window frame** because 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 port rectangle.

Window frame colors and patterns

In addition to the standard window types and controls, the color of the window and controls can be selected. Each 4-bit color is an index into either the default color table or a color table pointed to by the *wColor* field in the NewWindow parameter list (see Table 25-8 in the section "NewWindow" in this chapter). See Chapter 16, "QuickDraw II," for more information about color tables.

The color table for document and alert windows is shown in Figure 25-9.

Offset	Field	
$0 1	*frameColor*	**Word**—Color of window frame and alert frame Bits 15-8 = 0 Bits 7-4 = Outline color Bits 3-0 = 0
2 3	*titleColor*	**Word**—Color of inactive title bar, inactive title, and active title Bits 15-12 = 0 Bits 11-8 = Inactive title bar color Bits 7-4 = Inactive title color Bits 3-0 = Color of title, close, and zoom boxes
4 5	*tBarColor*	**Word**—Color and pattern of title bar Bits 15-8 = $0000 = Solid, $0001 = Dither, $0002 = Lined Bits 7-4 = Pattern color Bits 3-0 = Background color
6 7	*growColor*	**Word**—Color of size box and alert frame's middle outline Bits 15-12 = Color of alert frame's middle outline Bits 11-8 = 0 Bits 7-4 = Interior color of size box when not selected Bits 3-0 = Interior color of size box when selected
8 9	*infoColor*	**Word**—Color of information bar and alert frame's inside outline Bits 15-12 = Color of alert frame's inside outline Bits 11-8 = 0 7-4 = Interior color of information bar Bits 3-0 = 0

Figure 25-9
Document and alert window color table

Figures 25-10 through 25-14 show how these colors are used.

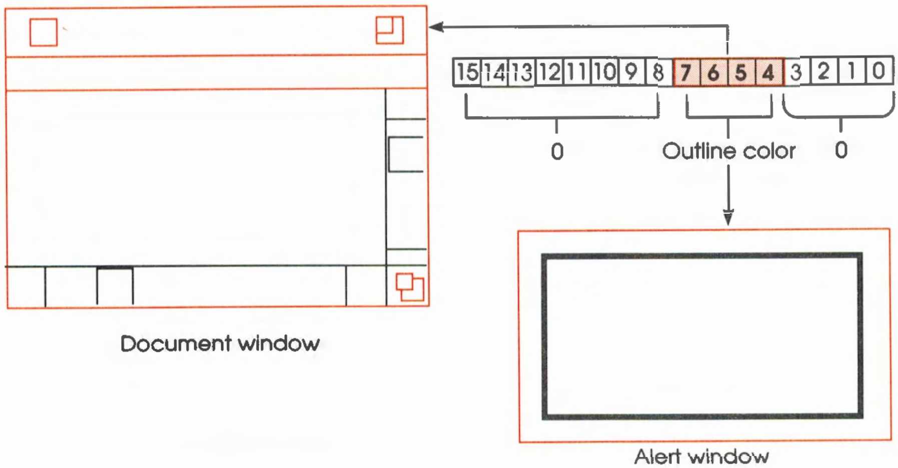


Figure 25-10
Window frame color (*frameColor*)

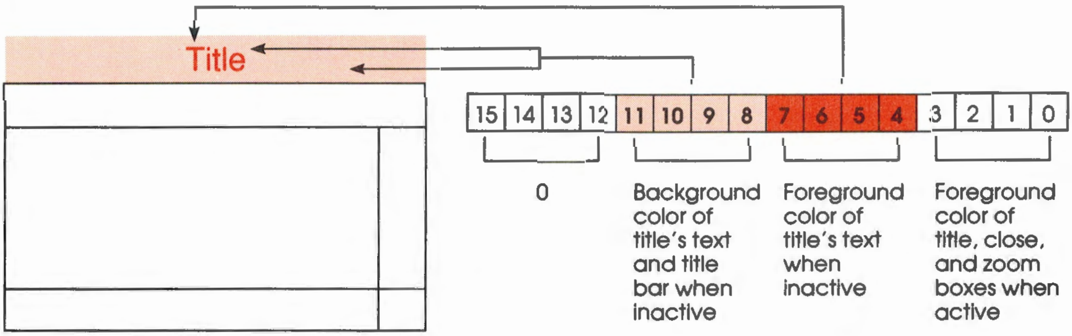


Figure 25-11
Window title color (*titleColor*)

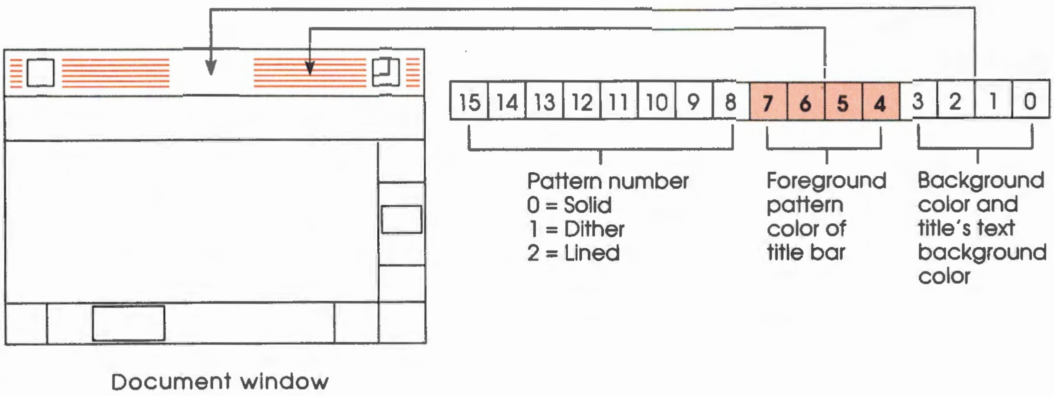


Figure 25-12
Window title bar color (*tBarColor*)

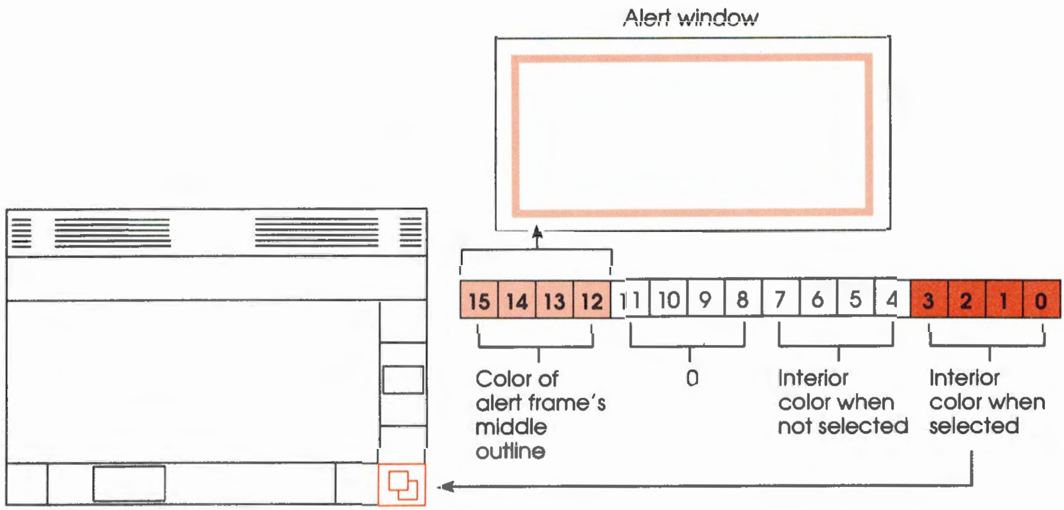


Figure 25-13
Window size box and alert window's middle outline color (*growColor*)

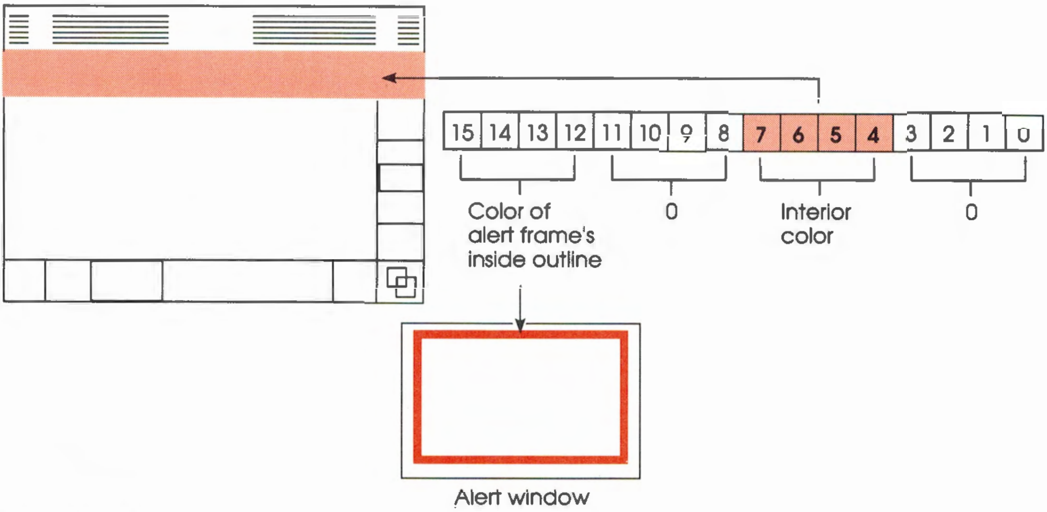


Figure 25-14
Window information bar and alert window's inside outline color (*infoColor*)

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

How a window is drawn

When a window is drawn or redrawn, the window frame is drawn first, followed by the window contents.

To draw the window frame, 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 procedure with a request that the window frame be drawn. The window definition procedure is either within the Window Manager or in the application for custom windows (see the section "Defining Your Own Windows" in this chapter).

To draw the window contents, the Window Manager generates an update event. Whenever your application receives an update event, the application should respond as follows:

1. Call the BeginUpdate routine. This routine temporarily replaces the visible region of the window's GrafPort with the intersection of the visible region and the update region. It then clears the update region for that window.
2. Draw the window contents.
3. Call the EndUpdate routine to restore the actual visible region.

Update events are issued for the frontmost window first and the hindmost last.

Draw content routine

When the NewWindow call is used to open a new window, the Window Manager checks the *wContDefProc* field. If that field is nonzero, the value is considered to be the address of a routine in your application that will draw the window's content region.

The *wContDefProc* field must be set if you want to use window frame scroll bars (which are the scroll bars TaskMaster creates). TaskMaster will scroll the content and call *wContDefProc* to update the uncovered area when the user performs a scrolling action. *wContDefProc* could be considered a control action procedure.

The *wContDefProc* field might be useful even if you are not using window frame scroll bars. TaskMaster can handle your update events if *wContDefProc* is set. TaskMaster will call BeginUpdate, *wContDefProc*, and EndUpdate.

There are no inputs or outputs to your draw content routine.

❖ *Note:* Use the QuickDraw II routine GetPort to obtain the current window pointer.

Draw what is needed in the content and perform an RTL to exit. Remember that the content will have already been erased using the window port's background pattern and that the visible region is set to the area needing to be redrawn.

Warning

Do not change ports or perform a QuickDraw II SetOrigin call while in your draw content routine.

Draw information bar routine

If the *fInfo* bit (bit 4) is set to 1 in the *wFrameBits* field of the NewWindow parameter list, the window will have an information bar that appears just above the content region. The width of the information bar is same as the width of the window, and the height of the information bar is specified by the *wInfoHeight* field in the NewWindow parameter list.

The Window Manager draws the information bar, but it is up to the application to draw any information inside the bar. Your application can do this by storing the address of a draw information bar routine in the *wInfoDefProc* field of the NewWindow parameter list (and you must also set the *fInfo* bit of *wFrame*). When the standard window frame definition procedure draws the empty information bar, it will also call the procedure pointed to by *wInfoDefProc*.

The inputs to your routine will be as follows:

Stack	
-- *infoBarPtr* --	**Long**—POINTER to RECT data structure specifying enclosing rectangle
-- *infoData* --	**Long**—*wInfoRefCon* value from NewWindow parameter list
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
RTL | *RTL*	**3 bytes**—RTL address
RTL | ← **SP**	

An assembly-language example of a draw information bar routine that prints a string looks like this:

```
InfoDefProc     START
theWindow       equu    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, to get
                pha                             ; 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, to
                pha                             ; 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
infoStrg    dc      i'8',c'MyTitle'
            END
```

The example takes some liberties, such as assuming the color and writing mode of the pen when the text is written. When entered, the current port is the Window Manager's. You may 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 the example takes is when the text is centered vertically. You should make QuickDraw calls to find font height, to find the InfoBar height, and then to actually center the text. You should always use InfoBar as offsets into the information bar interior because the height could be different from time to time.

Important

Do not change the current port's *clipRgn* or *visRgn* fields unless you save and restore the original value.

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; it is 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 generated—such as when there's only one window in the window list or when the active window is permanently discarded (because 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.
- Remove the highlighting or blinking cursor from text being edited when the window becomes inactive and restore it when the window becomes active.
- Enable or disable a menu or certain menu items 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, such as a round or hexagonal window; QuickDraw and the Window Manager allow you to define your own window shape.

To define your own type of window, you must write a **window definition procedure** that defines the appearance and behavior of the window. Then, when the Window Manager needs to do something, it calls your routine and not its own.

You pass the address to the NewWindow call in the *wFrameDefProc* parameter (see the section "NewWindow" in this chapter). The inputs to your routine are

Stack before call

previous contents	
varCode	**Word**—Operation to be performed
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
-- *params* --	**Long**—Parameters used by some messages
	← **SP**

Stack after call

previous contents	
-- *outcomeFlag* --	**Long**—Returned parameter flag
	← **SP**

The *varCode* parameters are shown in Table 25-4.

Table 25-4
The *varCode* parameters for custom windows

Value	Name	Description
0	`wDraw`	Draw window frame
1	`wHit`	Tell what region the cursor was in when the mouse button was pressed
2	`wCalcRgns`	Calculate *wStructRgn* and *wContRgn*
3	`wNew`	Do any additional window initialization
4	`wDispose`	Take any additional disposal actions
5	`wGrow`	Draw window's grow image

What you can expect in response to each of the codes is described immediately following the *wContDefProc* example.

Your routine must strip off the three input parameters and return via RTL. So the shell of your *wContDefProc* routine might be as follows:

```
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 window's frame
              dc      i2'test_hit-1'      ; Routine to find a window region
                                          ; at a given point
              dc      i2'calc_rgns-1'     ; Compute window's wStructRgn and wContRgn
              dc      i2'init_wind-1'     ; Do additional initialization
              dc      i2'kill_wind-1'     ; Do additional disposal
              END
draw_wind     START
              ...
              (code that draws window frame)
              jmp     exit
              END
test_hit      START
              ...
              (code that finds area of the window in which the point in params is located)
              jmp     exit
              END
calc_rgns     START
              ...
              (code that computes the window's wStructRgn and wContRgn)
              jmp     exit
              END
```

```
init_wind       START
                ...
                (code that performs additional initialization)
                jmp     exit
                END
kill_wind       START
                (code that performs 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
```

wDraw: draw a window frame

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.

The structure of the *params* parameter is as follows:

`wDrawFrame`	\$00	Draw window's entire frame.
`wInGoAway`	\$01	Draw go-away region.
`wInZoom`	\$02	Draw zoom region.
Bit 31	1	Draw frame or region as highlighted.
	0	Draw frame or region as unhighlighted.

Thus, valid *params* values are as follows:

\$00000000	Draw entire window frame as an inactive window.
\$80000000	Draw entire window frame as an active window.
\$00000001	Draw go-away region as unhighlighted.
\$80000001	Draw go-away region as highlighted.
\$00000002	Draw zoom region as unhighlighted.
\$80000002	Draw zoom region as highlighted.

wHit: find what region a point is in

The *params* parameter specifies 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 requests this operation only if the window is visible. Your routine should determine where the point is in your window and then return one of the following:

`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 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 in 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 a window's regions

Your routine should calculate the window's entire region, place it in the *wStrucRgn,* and place the content region in *wContRgn* based on the current GrafPort's port rectangle. The Window Manager requests this operation only if the window is visible.

Warning

When you calculate regions for your window, do not alter the *clipRgn* or *visRgn* fields of the window's GrafPort.

---

wNew: perform additional initialization

After initializing fields appropriately 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 may want to allocate your own record structure, initialize it, and store its pointer via a SetWRefCon call.

wDispose: remove a 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. The routine is called before all controls in the *wControls* and *wFrameCtrls* lists are removed via a KillControls call to the Control Manager, the port is closed, and the window record freed. Return 0 to continue closing or 1 to abort closing.

wGrow: draw the outline of a window

The *params* parameter is a pointer to a RECT data structure defining a rectangle. Your routine should draw an outline image of your window that would fit the specified 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.

Origin movement

This section describes in detail how the origin of a window can change and what the effects of that change are. To benefit from the following discussion, you should already be familiar with QuickDraw II's explanation of ports and boundsRect.

The origin of a window is what allows data to be scrolled and drawing to occur in the proper place after a scroll. In Figure 25-15, the gray area is a screen with the pixel in its upper left corner being 0,0 (coordinates are shown here as Y,X). The window port appears on the screen at 65,50 to 85,80. These points are called **global coordinates.** To draw the house, the X coordinate of the left side would be 60; that is, it would be 10 pixels inside the window port.

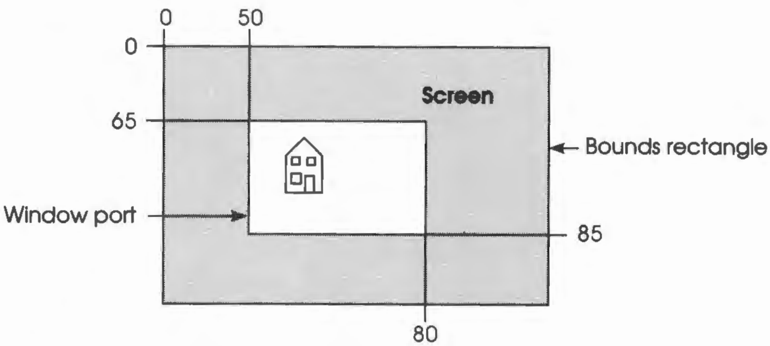


Figure 25-15
Window origin

However, a window port has its own coordinates, called **local coordinates.** Figure 25-16 shows what really happens when a window is created. Although the window is still at 65,50 to 85,80 on the screen, the local coordinates of the window are 0,0 to 20,30. Notice that the window's height and width remain the same. Also notice that the window is now called the port rectangle and the screen the bounds rectangle.

To draw the left wall of this house, you would pass the X coordinate of 10 to QuickDraw II to draw a single vertical line. QuickDraw II would then subtract the X origin of the boundsRect to determine where on the screen to actually draw the line. So, 10 minus –50 is 60. The global coordinate is 60; the local was 10. You will always work with local coordinates; that way, if the window is moved, its coordinate system remains the same. This explains what happens to the horizontal axis; the same thing happens with the vertical axis.

The Window Manager changes the coordinates of the bounds rectangle when the window is moved. If the window is moved one pixel to the left, the boundsRect would become –65,–49. When the coordinate of 10 is passed to QuickDraw II, it computes the global coordinate of 59 (10 minus –49). Thus, even though the house is drawn on another place on the screen, it is drawn in the same place in the window, and the application doesn't have to make any changes.

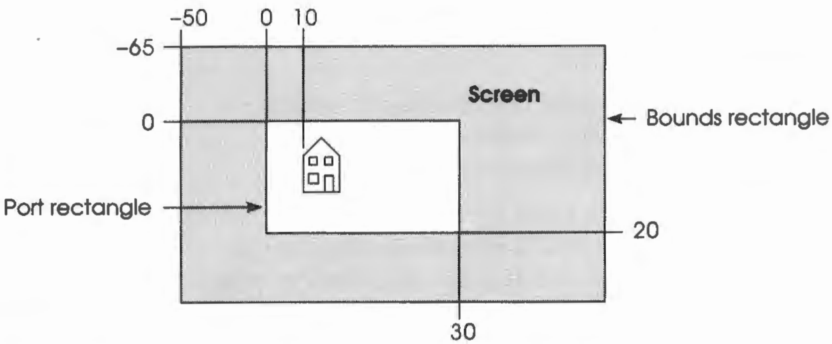


Figure 25-16
Window moving and origins

In the previous simple examples, everything had origins of 0,0. However, one of the powerful features of windows is their ability to scroll to show more data than the screen allows. In the next example, the window has not moved, but the user has scrolled the picture using the bottom scroll bar (see Figure 25-17). When the user moved the thumb on the scroll bar, the rectangle (0,10,20,30) was scrolled (moved) to (0,0,20,20). Then the origin of the window was changed to 0,10, and the exposed rectangle on the right side was redrawn in an update event.

After the scrolling occurred, the application would pass the coordinate of 10 to draw the left side of the house and QuickDraw II would compute 10 minus –40 to get the global coordinate of 50. Now, a minor problem arises: the Window Manager needs every window to have an origin of 0,0 for it to move, grow, and overlap windows. This feature may eventually change, but for now, whenever the Window Manager is called, the origin of every window must be 0,0.

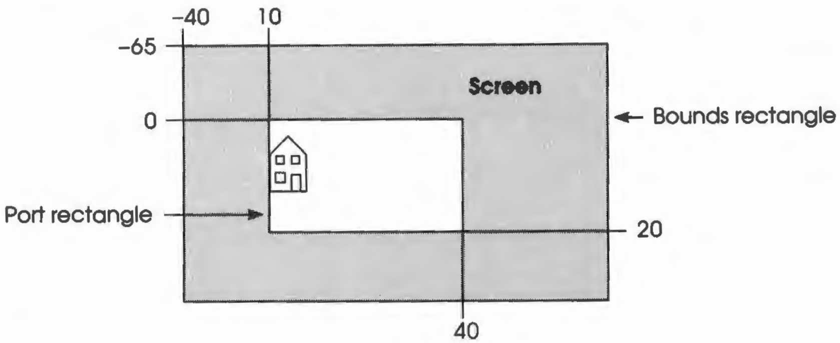


Figure 25-17
Scrolling and window origins

For example, whenever TaskMaster calls your update routine, it switches to the window's port and sets its origin; in this example it would be SetOrigin(0,10). Then you can draw in the window's local coordinates. When you have finished drawing and return to TaskMaster it performs a SetOrigin(0,0).

Even though changing the origin does not change the screen, any drawing outside of your update routine without setting the origin would have undesirable results. Drawing your house with the origin still at 0,0 would produce two houses, one shifted 10 pixels to the right of the other. To draw outside of your update routine, you need to first set the origin either yourself or through a StartDrawing call and then perform a SetOrigin(0,0) to put it back.

In short, when drawing outside of your update routine, you must perform a StartDrawing call before drawing and a SetOrigin(0,0) when you are finished drawing. This is also true when you are performing a hit test in your content region; the event position must be converted to local coordinates.

$010E WindBootInit

Initializes the Window Manager; called only by the Tool Locator.

Warning
An application must never make this call.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C Call must not be made by an application.

$020E WindStartUp

Starts up the Window Manager for use by an application.

Important
Your application must make this call before it makes any other Window Manager calls.

Parameters

Stack before call

previous contents
userID — **Word**—ID number of application
← **SP**

Stack after call

previous contents
← **SP**

Errors None

C

```
extern pascal void WindStartUp(userID)

Word     userID;
```

$030E WindShutDown

Shuts down the Window Manager down when an application quits.

Important
If your application has started up the Window Manager, the application must make this call before it quits.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

None

C

```
extern pascal void WindShutDown()
```

$040E WindVersion

Returns the version number of the Window Manager.

Parameters

Stack before call

Stack	
previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

Stack	
previous contents	
versionInfo	**Word**—Version number of Window Manager
	← **SP**

Errors

None

C

```
extern pascal Word WindVersion()
```

$050E WindReset

Resets the Window Manager; called only when the system is reset.

Warning
An application must never make this call.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C Call must not be made by an application.

$060E WindStatus

Indicates whether the Window Manager is active.

Parameters

Stack before call

Stack	
previous contents	
wordspace	**Word**—Space for result
	← **SP**

Stack after call

Stack	
previous contents	
activeFlag	**Word**—BOOLEAN; TRUE if Window Manager active, FALSE if inactive
	← **SP**

Errors None

C

```
extern pascal Boolean WindStatus()
```

$1E0E BeginUpdate

Replaces the visible region of the window's GrafPort with the intersection of the visible region and the update region and then sets the window's update region to an empty region.

Call BeginUpdate when an update event occurs for the window. You would then usually draw the entire content region, although it suffices to draw only the visible region. In either case, only the parts of the window that require updating and are visible will actually be drawn on the screen. Every call to BeginUpdate must be balanced by a call to the EndUpdate routine, as follows:

1. Call BeginUpdate.
2. Draw the window contents.
3. Call the EndUpdate routine to restore the actual visible region.

Parameters

Stack before call

previous contents	
— *theWindowPtr* —	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void BeginUpdate(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$240E BringToFront

Brings a specified window to the front of all other windows and redraws the windows as necessary but does not do any highlighting or unhighlighting. Normally you won't have to call this procedure; you should call SelectWindow to make a window active, and SelectWindow takes care of bringing the window to the front. If you do call BringToFront, however, remember to call HiliteWindow to make any necessary highlighting changes.

Parameters

Stack before call

previous contents
-- *theWindowPtr* --
← **SP**

theWindowPtr: **Long**—POINTER to window's GrafPort

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void BringToFront(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$0A0E CheckUpdate

Looks from top to bottom in the window list for a visible window that needs updating (that is, for a window whose update region is not empty). CheckUpdate is normally called only by the Event Manager, and doesn't need to be called by an application.

If a window with something in its update region is found, an update event for that window is stored in the event and the routine returns TRUE. If it doesn't find such a window, it returns FALSE.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *theEventPtr* --	**Long**—POINTER to an event record
	← **SP**

Stack after call

previous contents	
updateFlag	**Word**—BOOLEAN; TRUE if update event found, otherwise FALSE
	← **SP**

Errors

None

C

```
extern pascal Boolean CheckUpdate(theEventPtr)

EventRecordPtr     theEventPtr;
```

$0B0E CloseWindow

Removes a specified window from the screen, disposes of all controls associated with that window, and deletes the window from the window list. The routine releases the memory occupied by all data structures associated with the window, including the memory taken up by the window record if it was allocated by NewWindow. Call this routine when you're done with a window.

Any update events for the window are discarded. If the deleted window was the frontmost window, the window behind it (if any) is highlighted and an appropriate activate event is generated.

Warning
If you allocated memory yourself and stored a handle to it in the *wRefCon* field, CloseWindow won't know about it—you must release the memory before calling CloseWindow.

Parameters

Stack before call

Stack	
previous contents	
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors None

C

```
extern pascal void CloseWindow(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$0C0E Desktop

Controls the addition of regions to and subtraction of regions from the desktop and controls the current desktop pattern. The values for the *deskTopOp* and *dtParam* parameters are shown in Table 25-5.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result if necessary
deskTopOP	**Word**—Operation to perform
-- *dtParam* --	**Long**—Parameter needed for operation
	← **SP**

Stack after call

previous contents	
-- *retParam* --	**Long**—Possible return parameter (see Table 25-5)
	← **SP**

Errors

None

C

```
extern pascal Pointer Desktop(deskTopOP,dtParam)

Word      deskTopOP;

LongWord      dtParam;
```

(continued)

Table 25-5
Window Manager Desktop routine operations and parameters

Operation	Operation number	Description and *dtParam* and *retParam* values
`FromDesk`	0	Subtract region from desktop region. *dtParam*—HANDLE to region to be subtracted *retParam*—Not used The region passed in *dtParam* will be subtracted from the current desktop region. When the Window Manager redraws the desktop, it will not draw the region subtracted. Windows will not cover the subtracted region and will appear to move underneath the region. For example, the Menu Manager subtracts a region from the desktop for the system menu bar, which is why windows move under the system menu bar. `FromDesk` can be called repeatedly to remove additional areas from the desktop.
`ToDesk`	1	Add region to current desktop region. *dtParam*—HANDLE to region to be added *retParam*—Not used When the Window Manager redraws the desktop, it will also draw the region added. `ToDesk` can be called repeatedly to add additional areas to the desktop.
`GetDesktop`	2	Return handle to current desktop region. *dtParam*—Not used *retParam*—HANDLE to desktop region The handle returned is the actual handle to the desktop region; any modifications to this region will change the desktop. Using the handle, you can add, subtract, or XOR a region; find intersections; and perform other region operations. This call can also be used for restoring the desktop to its original shape after modifying it for some temporary use (see `SetDesktop` for restoring the desktop). **Warning** Do not free the handle of the desktop region; that will cause the Window Manager to crash.

Table 25-5 (continued)
Window Manager Desktop routine operations and parameters

Operation	Operation number	Description and *dtParam* and *retParam* values
`SetDesktop`	3	Set handle to desktop region. *dtParam*—HANDLE to new desktop region *retParam*—Same as *param* After `SetDesktop` is called, the new handle will be used by the Window Manager for the handle of the desktop region. It is not necessary to call `SetDesktop` if you have called `GetDesktop` and modified the region because `GetDesktop` returns the actual desktop handle. **Warning** SetDesktop overwrites the current desktop region handle. Therefore, it is up to the application to free the original handle or save it and restore it later.
`GetDeskPat`	4	Return current desktop pattern. *dtParam*—Not used *retParam*—Current desktop pattern (see Table 25-6) `GetDeskPat` returns information about how the desktop is being drawn. There are several ways the desktop can be drawn, as shown in Table 25-6.
`SetDeskPat`	5	Set new desktop pattern. The desktop is redrawn with the new pattern. *dtParam*—New desktop pattern (see Table 25-6) *retParam*—Not used `SetDeskPat` changes how the desktop is drawn. There are several ways the desktop can be drawn, as specified in Table 25-6.
`GetVisDesktop`	6	Return desktop, less any windows. *dtParam*—Handle to region that will be set to the visible desktop *retParam*—Not used The current desktop region minus any visible windows is copied into the given region. The visible desktop can be used for drawing on the desktop. See the following discussion of `BackGroundRgn` for an extension to `GetVisDesktop`.

(continued)

Table 25-5 (continued)
Window Manager Desktop routine operations and parameters

Operation	Operation number	Description and *dtParam* and *retParam* values
`BackGroundRgn`	7	Maintain visible desktop region. *dtParam*—Handle to a region *retParam*—Not used The region passed will be set to the desktop region less any windows. The region is automatically updated when windows are added, removed, sized, and moved. This operation provides applications an easy way of drawing objects directly on the desktop. A possible sequence for drawing objects on the desktop might be as follows:

```
pha                                  ; Space for result (not used)
pha
pea     SetDeskPat                   ; Operation number 5
pea     MyDeskDraw|-16               ; Pass address of my routine
pea     MyDeskDraw                   ; that will draw the desktop
_Desktop
pla                                  ; Result not used
pla
pea     MyDeskPort|-16               ; Open a port for my desktop
pea     MyDeskPort
_OpenPort
pha                                  ; Space for result (not used)
pha
pea     BackGroundRgn                ; Operation number 7
lda     MyDeskPort+VisRgn+2          ; Pass handle of my desktop
                                     ; port's visRgn
pha
lda     MyDeskPort+VisRgn
pha
_Desktop
pla                                  ; Result not used
pla
```

Table 25-5 (continued)
Window Manager Desktop routine operations and parameters

Operation	Operation number	Description and *dlParam* and *retParam* values
		After this code is executed, the routine `MyDeskDraw` will be called by the Window Manager whenever the desktop needs to be drawn. The preceding code passed the value of the *visRgn* field of a port to the Window Manager. The Window Manager will use that value to compute the visible desktop. Then, when an application wants to draw an object on the desktop, it can switch to `MyDeskPort` and draw. All drawing will be clipped to the current visible desktop. ❖ *Note:* The address of the routine passed to the Desktop routine for operation `SetDeskPat` (operation 5) is still a routine that is called with the current port being the Window Manager's with its clip region set to the visible desktop. To draw whenever you want, you'll have to use your own port and your own visible region.

Desktop patterns

There are no inputs to or outputs from the routine called to draw the desktop. The current port will be the Window Manager's, and the clipping region will be set to the area needing to be drawn. Your routine should exit via an RTL.

Warning
The current direct page and data bank are not defined on entry to your routine. When you exit your routine, the direct page and data bank must be the same as they were on entry.

The desktop pattern is determined by a long value, as shown in Table 25-6.

Table 25-6
Desktop patterns

Byte 1	Byte 2	Byte 3	Byte 4
\$00	Address of routine that will be called to draw desktop		
\$80	Address of pattern to be used for desktop (see Chapter 16, "QuickDraw II")		
\$40	00	00 = Solid desktop pattern 01 = Dithered desktop pattern 02 = Horizontally striped desktop pattern	High nibble = Pattern's foreground color Low nibble = Pattern's background color

$1A0E DragWindow

Pulls around a dotted outline of a specified window, following the movements of the mouse until the mouse button is released. When the button is released, DragWindow calls MoveWindow to move the specified window to the location to which it was dragged. The window will be dragged and moved in its current plane.

When there is a mouse-down event in the drag region of the specified window and TaskMaster is not being used, the application should call DragWindow with *startY*, *startX* equal to the point where the mouse button was pressed (in global coordinates, as stored in the *where* field of the event record).

Parameters

Stack before call

previous contents	
grid	**Word**—Drag resolution, zero for default (see Table 25-7)
startX	**Word**—Starting X coordinate of cursor, in global coordinates
startY	**Word**—Starting Y coordinate of cursor, in global coordinates
grace	**Word**—Grace buffer around bounds
-- *boundsRectPtr* --	**Long**—POINTER to RECT structure for cursor boundary, NIL for default
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void DragWindow(grid,startX,startY,grace,boundsRectPtr,
theWindowPtr)

Word      grid;

Integer     startX;

Integer     startY;

Word      grace;

Rect *boundsRectPtr;

GrafPortPtr     theWindowPtr;
```

You can also use the following alternate form of the call:

```
extern pascal void DragWindow(grid,start,grace,boundsRectPtr,theWindowPtr)
Word     grid;
Point     start;
Word     grace;
Rect *boundsRectPtr;
GrafPortPtr     theWindowPtr;
```

Parameter description

grid — The allowed horizontal resolution movement, as shown in Table 25-7. The *grid* parameter is provided to speed up window moves by eliminating the need for bit shifting, if the *grid* value is the correct value. If *grid* is passed as 0, a default value will be used. The defaults are 4 for 320 mode and 8 for 640 mode. The only allowed values are 1, 2, 4, 8, 16, 32, 64, 128, and so on.

Table 25-7
DragWindow grid values

Value	Window movement
0	Default value used.
1	Window can be positioned at any horizontal position.
2	Window can only be moved a multiple of 2 pixels horizontally.
4	Window can only be moved a multiple of 4 pixels horizontally.
8	Window can only be moved a multiple of 8 pixels horizontally.

startY and *startX* — Indicate where the mouse button was pressed, in global coordinates, as stored in the *where* field of the event record. This point is used with the tracked cursor position to compute the movement delta.

(continued)

grace The distance, in pixels, that you will allow the cursor to move away from the bounds rectangle before the dragged outline should be snapped back to its starting position. TaskMaster uses 8 for this value. The bounds rectangle is expanded by the value of grace to compute the slop rectangle that is passed to DragRect as the *slopRect* parameter. See the section "DragRect" in Chapter 4, "Control Manager," in Volume 1 for more information.

boundsRectPtr Pointer to a RECT data structure, in global coordinates, that is passed to DragRect as the *limitRect* parameter. See the section "DragRect" in Chapter 4, "Control Manager," in Volume 1 for more information. If NIL is passed for the pointer, the bounds of the desktop, minus 4 all around, will be used.

$510E EndInfoDrawing

Puts the Window Manager back into a global coordinate system. Call this routine after a StartInfoDrawing call and before any other calls to the Window Manager.

Warning

Calling any Window Manager routine between a StartInfoDrawing call and an EndInfoDrawing call may result in system failure.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

```
extern pascal void EndInfoDrawing()
```

$1F0E EndUpdate

Restores the normal visible region of a specifed window's GrafPort that was changed by a BeginUpdate call.

Parameters

Stack before call

previous contents	
— *theWindowPtr* —	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors None

C

```
extern pascal void EndUpdate(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$170E FindWindow

Tells which part of which window, if any, the cursor was in when the user pressed the mouse button. If it was pressed in a window, the *whichWindowPtr* parameter is set to the window port pointer; otherwise, it's set to NIL.

When a mouse-down event occurs, the application should, if not using TaskMaster, call FindWindow with *pointX* and *pointY* equal to the point where the cursor was when the user pressed the mouse button (in global coordinates, as stored in the *where* field of the event record).

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *whichWindowPtr* --	**Long**—POINTER to space to store pointer to window; NIL if not found
pointX	**Word**—X point to check, in global coordinates
pointY	**Word**—Y point to check, in global coordinates
	← **SP**

Stack after call

previous contents	
location	**Word**—Mouse-down event location (see Table 25-8)
	← **SP**

Errors

None

C

```
extern pascal Word FindWindow(whichWindowPtr,pointX,pointY)

GrafPortPtr *whichWindowPtr;

Integer     pointX;

Integer     pointY;
```

You can also use the following alternate form of the call:

```
extern pascal Word FindWindow(whichWindowPtr,point)

GrafPortPtr *whichWindowPtr;

Point     point;
```

Mouse-down event location information

The *location* returned by FindWindow is one of the constants shown in Table 25-8.

Table 25-8
FindWindow constants

Word	Event	Description
$0000	`wNoHit`	Not in the window at all
$0010	`wInDesk`	In the desktop area
$0011	`wInMenuBar`	In the system menu bar
$0013	`wInContent`	In window's content region
$0014	`wInDrag`	In window's drag (title bar) region
$0015	`wInGrow`	In window's size box region
$0016	`wInGoAway`	In window's go-away (close box) region
$0017	`wInZoom`	In window's zoom (zoom box) region
$0018	`wInInfo`	In window's information bar
$0019	`wInSpecial`	In special menu item bar (see Chapter 13, "Menu Manager," in Volume 1)
$001A	`wInDeskItem`	Desk accessory selected from the Apple menu
$001B	`wInFrame`	In window, but not in any of the areas defined in this table
$001C	`wInactMenu`	Inactive menu item selected
$8xxx	`wInSysWindow`	In a system window

$150E FrontWindow

Returns a pointer to the first visible window in the window list (that is, the active window). If there are no visible windows, the routine returns NIL.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
	← **SP**

Stack after call

previous contents	
-- *theWindowPtr* --	**Long**—POINTER to active window's GrafPort; NIL if none visible
	← **SP**

Errors

None

C

```
extern pascal GrafPortPtr FrontWindow()
```

$480E GetContentDraw

Returns the pointer to the routine that draws the content region of a specified window.

TaskMaster calls this routine when it gets an update event for that window. See the section "Draw Content Routine" in this chapter for more information about the draw routine.

Parameters

Stack before call

previous contents	
— *longspace* —	**Long**—Space for result
— *theWindowPtr* —	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
— *contentProcPtr* —	**Long**—POINTER to routine called to draw content region
	← **SP**

Errors

None

C

```
extern pascal VoidProcPtr GetContentDraw(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$3E0E GetContentOrgin

Returns the values used by TaskMaster to set the origin of the window's GrafPort when handling an update event. The values are also used to compute scroll bars in the window frame.

Parameters

Stack before call

previous contents	
— *longspace* —	**Long**—Space for result
— *theWindowPtr* —	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
— *origin* —	**Long**—POINT; low word = Y origin, high word = X origin
	← **SP**

Errors

None

C

```
extern pascal Long GetContentOrigin(theWindowPtr)
```

```
Pointer     theWindowPtr;
```

❖ *Note:* C Pascal-type functions do not deal properly with data structures returned on the stack. The Long result returned by this call can be passed to any calls requiring a point as a parameter. You cannot use the C dot operator to access the individual Y and X coordinates within the value returned by this call.

$2F0E GetContentRgn

Returns a handle to a specified window's content region. See the section "Window Regions" in this chapter for a definition of the content region.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
-- *wContHandle* --	**Long**—HANDLE to window's content region
	← **SP**

Errors

None

C

```
extern pascal RgnHandle GetContentRgn(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$400E GetDataSize

Returns the height and width of the data area of a specified window. The data area is the total amount of data that can be viewed in a window through resizing or scrolling.

Parameters

Stack before call

previous contents
-- *longspace* --
-- *theWindowPtr* --

longspace: **Long**—Space for result

theWindowPtr: **Long**—POINTER to window's GrafPort

← **SP**

Stack after call

previous contents
-- *dataSize* --

dataSize: **Long**—Low word = height; high word = width

← **SP**

Errors

None

C

```
extern pascal LongWord GetDataSize(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$310E GetDefProc

Returns a pointer to the routine that is called to draw, hit test, and otherwise define a window's frame and behavior.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
-- *wDefProcPtr* --	**Long**—POINTER to window's definition procedure
	← **SP**

Errors

None

C

```
extern pascal LongProcPtr GetDefProc(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$520E GetFirstWindow

Returns a pointer to the first window in the Window Manger's window list. The returned window may not be the active window (see the section "FrontWindow" in this chapter). Every window in the window list, whether visible or not, can be accessed if you call GetFirstWindow and then call the GetNextWindow routine to run down the remainder of the window list.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
	← **SP**

Stack after call

previous contents	
-- *firstWindowPtr* --	**Long**—POINTER to first window, or NIL
	← **SP**

Errors

None

C

```
extern pascal GrafPortPtr GetFirstWindow()
```

$100E GetFrameColor

Returns the color of a specified window's frame. See the section "Window Frame Colors and Patterns" in this chapter for a definition of the color table.

Parameters

Stack before call

previous contents	
-- *colorPtr* --	**Long**—POINTER to five-word table to be filled with window's color table
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void GetFrameColor(colorPtr,theWindowPtr)

WindColorPtr      colorPtr;

GrafPortPtr     theWindowPtr;
```

$4A0E GetInfoDraw

Returns the pointer to the routine that draws the information bar for a specified window. If the window has an information bar routine, the standard window definition procedure calls that routine whenever the window's frame needs to be drawn. See the section "Draw Information Bar Routine" in this chapter for more information about that routine.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
-- *infoDrawPtr* --	**Long**—POINTER to routine to draw infomation bar
	← **SP**

Errors

None

C

```
extern pascal VoidProcPtr GetInfoDraw(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$350E GetInfoRefCon

Returns the value of a specified window's *wInfoRefCon* field (the value associated with the draw information bar routine). The field is reserved for application use.

Parameters

Stack before call

previous contents	
longspace	**Long**—Space for result
theWindowPtr	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
infoRefCon	**Long**—Value passed to draw information bar routine
	← **SP**

Errors

None

C

```
extern pascal LongWord GetInfoRefCon(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$420E GetMaxGrow

Returns the maximum values to which a specified window's content region can grow.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
-- *maxGrow* --	**Long**—Low word = *maxHeight;* high word = *maxWidth*
	← **SP**

Errors

None

C

```
extern pascal LongWord GetMaxGrow(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$2A0E GetNextWindow

Returns a pointer to the next window in the window list after a specified window; returns NIL if the specified window is the last window in the window list.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
-- *nextWindowPtr* --	**Long**—POINTER to next window's GrafPort in list; NIL if last window
	← **SP**

Errors

None

C

```
extern pascal GrafPortPtr GetNextWindow(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$460E GetPage

Returns the number of pixels by which TaskMaster will scroll the content region when the user selects the page regions on window frame scroll bars.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
-- *pageAmount* --	**Long**—Low word = vertical amount; high word = horizontal amount
	← **SP**

Errors

None

C

```
extern pascal LongWord GetPage(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$4F0E GetRectInfo

Sets the information rectangle to the coordinates of the information bar rectangle. If there is no information bar in the specified window, the coordinates of the RECT data structure pointed to by *infoRectPtr* will all be 0. The coordinate system will be local to the window's frame; that is, 0,0 will be the upper left corner of the window. The coordinates can be used to set the position of objects that will be drawn in the information bar.

Parameters

Stack before call

previous contents	
-- *infoRectPtr* --	**Long**—POINTER to destination RECT where rectangle will be stored
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void GetRectInfo(infoRectPtr,theWindowPtr)

Rect *infoRectPtr;

GrafPortPtr     theWindowPtr;
```

$440E GetScroll

Returns the number of pixels by which TaskMaster will scroll the content region when the user selects the arrows on window frame scroll bars.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
-- *scrollAmount* --	**Long**—Low word = vertical amount; high word = horizontal amount
	← **SP**

Errors

None

C

```
extern pascal LongWord GetScroll(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$2E0E GetStructRgn

Returns a handle to a specified window's structure region. See the section "Window Regions" in this chapter for a definition of the structure region.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
-- *wStructHandle* --	**Long**—HANDLE to window's structure region
	← **SP**

Errors

None

C

```
extern pascal RgnHandle GetStructRgn(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$4C0E GetSysWFlag

Indicates whether a specified window is a system window or an application window.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
sysFlag	**Word**—BOOLEAN; TRUE if system window, FALSE if application window
	← **SP**

Errors

None

C

```
extern pascal Boolean GetSysWFlag(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$300E GetUpdateRgn

Returns a handle to a specified window's update region. See the section "BeginUpdate" in this chapter for an explanation of how the update region is used.

Parameters

Stack before call

previous contents	
— *longspace* —	**Long**—Space for result
— *theWindowPtr* —	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
— *wUpdateHandle* —	**Long**—HANDLE to window's update region
	← **SP**

Errors

None

C

```
extern pascal RgnHandle GetUpdateRgn(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$330E GetWControls

Returns the handle to the first control in the window's control list. The window's control list is the list of controls created by the application with calls to NewControl in the Control Manager.

Parameters

Stack before call

previous contents	
— *longspace* —	**Long**—Space for result
— *theWindowPtr* —	**Long—**POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
— *controlHandle* —	**Long—**HANDLE to first control in window's control list; NIL if none
	← **SP**

Errors

None

C

```
extern pascal CtlRecHndl GetWControls(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$2C0E GetWFrame

Returns the bit flag that describes a specified window's frame type. See the discussion of the *wFrame* field in the section "NewWindow" in this chapter for the definition of the bits in the flag.

Parameters

Stack before call

Stack	
previous contents	
wordspace	**Word**—Space for result
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

Stack	
previous contents	
wFrame	**Word**—Bit flag with window's frame type
	← **SP**

Errors

None

C

```
extern pascal Word GetWFrame(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$2B0E GetWKind

Indicates whether a specified window is a system window or an application window.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
windowKind	**Word**—Bit 15 is 0 for application window, 1 for system window
	← **SP**

Errors

None

C

```
extern pascal Word GetWKind(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$200E GetWMgrPort

Returns a pointer to the Window Manager's port.

Parameters

Stack before call

previous contents	
longspace	**Long**—Space for result
	← **SP**

Stack after call

previous contents	
wPortPtr	**Long**—POINTER to Window Manager's GrafPort
	← **SP**

Errors

None

C

```
extern pascal GrafPortPtr GetWMgrPort()
```

$290E GetWRefCon

Returns a value from a specified window's record that was passed to either NewWindow or SetWRefCon by the application. The *wRefCon* field is reserved for use by the application.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
-- *wRefCon* --	**Long**—Current value of *wRefCon* field
	← **SP**

Errors

None

C

```
extern pascal LongWord GetWRefCon(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$0E0E GetWTitle

Returns the pointer to a specified window's title. The string pointed to by *titlePtr* is a Pascal-type string.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
-- *titlePtr* --	**Long**—POINTER to window's title string
	← **SP**

Errors

None

C

```
extern pascal Pointer GetWTitle(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$370E GetZoomRect

Returns a pointer to the rectangle to be used as the content's zoomed or unzoomed size for a specified window. If the zoom flag is set in the frame flag (see the section "GetWFrame" in this chapter), then *wZoomSizePtr* points to a RECT data structure that contains the window's last size and position. Otherwise, *wZoomSizePtr* points to a RECT data structure that contains the size and position of the window's content region (port) the next time the window is zoomed by a call to ZoomWindow.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
← **SP**	

Stack after call

previous contents	
-- *wZoomSizePtr* --	**Long**—POINTER to RECT data structure used as content's zoomed size
← **SP**	

Errors

None

C

```
extern pascal Rect * GetZoomRect(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$1B0E GrowWindow

Pulls around a grow image of a specified window, following the movements of the mouse until the mouse button is released. The **grow image** for a document window is a dotted outline of the entire window plus the lines delimiting the title bar, size box, and scroll bar areas. See Figure 25-18.

When there's a mouse-down event in the size box region of the specified window, the application should call GrowWindow with *startY* and *startX* equal to the point where the mouse button was pressed (in global coordinates, as stored in the *where* field of the event record).

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
minWidth	**Word**—Minimum width of content region
minHeight	**Word**—Minimum height of content region
startX	**Word**—Starting X coordinate of cursor, in global coordinates
startY	**Word**—Starting Y coordinate of cursor, in global coordinates
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
-- *newSize* --	**Long**—High word = new width; low word = new height
	← **SP**

Errors

None

C

```
extern pascal LongWord GrowWindow(minWidth,minHeight,startX,startY,
theWindowPtr)
Word      minWidth;
Word      minHeight;
Integer      startX;
Integer      startY;
GrafPortPtr      theWindowPtr;
```

You can also use the following alternate form of the call:

```
extern pascal LongWord GrowWindow(minWidth,minHeight,start,theWindowPtr)
Word      minWidth;
Word      minHeight;
Point      start;
GrafPortPtr      theWindowPtr;
```

Grow image

Figure 25-18 illustrates the grow image for a document window that contains both scroll bars. In general, the grow image is defined in the window definition function to appropriately show that the window's size will change.

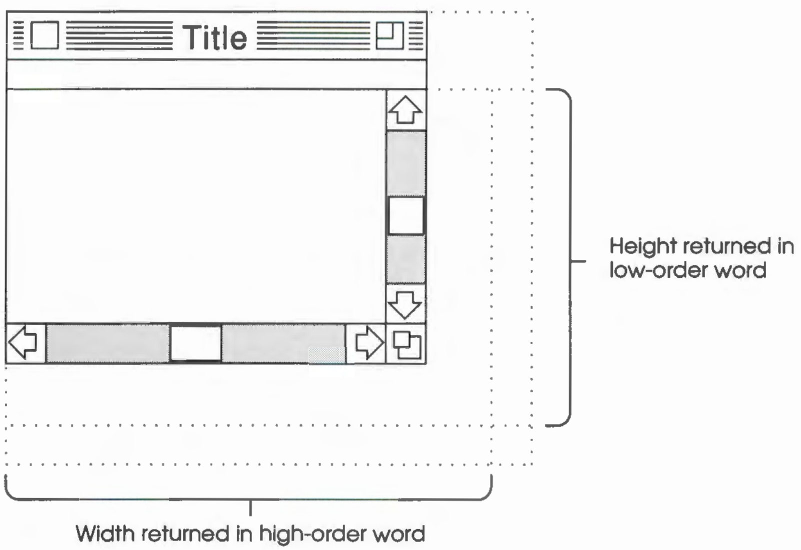


Figure 25-18
Grow image of a window

Your application should subsequently call the SizeWindow routine to change the port rectangle of the window's GrafPort to the new one outlined by the grow image. The *sizeRect* parameter specifies limits, in pixels, on the height (vertical measurement) and width (horizontal measurement) of what will be the new port rectangle. The top coordinate of *sizeRect* is the minimum vertical measurement, the left coordinate is the minimum horizontal measurement, the bottom coordinate is the maximum vertical measurement, and the right coordinate is the maximum horizontal measurement.

GrowWindow returns the actual size for the new port rectangle as outlined by the grow image when the mouse button is released. The high-order word of the long is the horizontal measurement in pixels; the low-order word is the vertical measurement. A return value of 0 indicates that the size is the same as that of the current port rectangle.

$120E HideWindow

Makes a specified window invisible. If the window is the frontmost window and there's a window behind it, HideWindow also unhighlights the window, brings the window behind it to the front, highlights that window, and generates appropriate activate events. If the specified window is already invisible, HideWindow has no effect.

Parameters

Stack before call

previous contents
-- *theWindowPtr* --
← **SP**

theWindowPtr: **Long**—POINTER to window's GrafPort

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void HideWindow(theWindowPtr)
GrafPortPtr     theWindowPtr;
```

$220E HiliteWindow

Highlights or unhighlights a specified window, depending on the value of a specified parameter. If *fHiliteFlag* is TRUE, this routine highlights the window. If *fHiliteFlag* is FALSE, HiliteWindow unhighlights the window. The exact way a window is highlighted and unhighlighted depends on its window definition procedure.

Normally you won't have to call this routine because you should call SelectWindow to make a window active and SelectWindow takes care of the necessary highlighting changes. To conform with the Apple *Human Interface Guidelines,* don't highlight a window that isn't the active window.

Parameters

Stack before call

previous contents	
fHiliteFlag	**Word**—BOOLEAN; TRUE to highlight window, FALSE to unhighlight
— *theWindowPtr* —	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void HiliteWindow(fHiliteFlag,theWindowPtr)

Boolean      fHiliteFlag;

GrafPortPtr     theWindowPtr;
```

$3A0E InvalRect

Accumulates a specified rectangle into the update region of the window whose GrafPort is the current port. This tells the Window Manager that the rectangle has changed and must be updated. The rectangle is given in local coordinates and is then clipped by the Window Manager to the window's content region.

Important

This routine changes the coordinates you give it. Save the coordinates if you need to restore them later.

Parameters

Stack before call

previous contents
-- *badRectPtr* --
← **SP**

badRectPtr: **Long**—POINTER to RECT data structure of rectangle to be added

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void InvalRect(badRectPtr)

Rect *badRectPtr;
```

$3B0E InvalRgn

Accumulates a specified region into the update region of the window whose GrafPort is the current port. This tells the Window Manager that the region has changed and must be updated. The region is given in local coordinates and is then clipped by the Window Manager to the window's content region.

Important

This routine changes the coordinates you give it. Save the coordinates if you need to restore them later.

Parameters

Stack before call

previous contents
-- *badRgnHandle* --
← **SP**

badRgnHandle: **Long**—HANDLE to region to be added to update region

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void InvalRgn(badRgnHandle)

RgnHandle     badRgnHandle;
```

$190E MoveWindow

Moves a specified window to another part of the screen without affecting the window's size. The upper left corner of the window's port rectangle is moved to the screen point *newY, newX*. The local coordinates of the window's top left corner remain the same.

Parameters

Stack before call

previous contents	
newX	**Word**—New X coordinate of content region's upper left corner (global)
newY	**Word**—New Y coordinate of content region's upper left corner (global)
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void MoveWindow(newX,newY,theWindowPtr)

Integer     newX;

Integer     newY;

GrafPortPtr     theWindowPtr;
```

You can also use the following alternate form of the call:

```
extern pascal void MoveWindow(newPoint,theWindowPtr)

Point     newPoint;

GrafPortPtr     theWindowPtr;
```

$090E NewWindow

Creates a specified window as specified by its parameters, adds it to the window list, and returns a pointer to the new window's GrafPort. NewWindow allocates space for the structure and content regions of the window and asks the window definition function to calculate those regions.

Important

NewWindow does not set the current port, but many routines require that a current port exist. Use the QuickDraw II routine SetPort to set the current port.

Parameters

Stack before call

previous contents	
-- *longspace* --	**Long**—Space for result
-- *paramListPtr* --	**Long**—POINTER to parameter list (see Table 25-9)
← **SP**	

Stack after call

previous contents	
-- *theWindow* --	**Long**—POINTER to window's GrafPort; NIL if error
← **SP**	

Errors

$0E01	`paramLenErr`	First word of parameter list is the wrong size
$0E02	`allocateErr`	Unable to allocate memory for window record

C

```
extern pascal GrafPortPtr NewWindow(paramListPtr)
ParamListPtr     paramListPtr;
```

(continued)

NewWindow parameter list

The NewWindow parameter list is shown, and each parameter is briefly described, in Table 25-9.

Table 25-9
NewWindow parameter list

Parameter	Length	Description
paramLength	Word	Total number of bytes in parameter table, including the *paramLength* parameter itself
wFrameBits	Word	Bit flag that describes window frame type
wTitle	Long	Pointer to window's title
wRefCon	Long	Reserved for application use
wZoom	4 words	RECT specifying size and position of content when zoomed
wColor	Long	Pointer to window's color table
wYOrigin	Word	Vertical offset of content region from data area
wXOrigin	Word	Horizontal offset of content region from data area
wDataH	Word	Height of entire document
wDataW	Word	Width of entire document
wMaxH	Word	Maximum height of content allowed by GrowWindow
wMaxW	Word	Maximum width of content allowed by GrowWindow
wScrollVer	Word	Number of pixels to scroll content vertically when user clicks arrow
wScrollHor	Word	Number of pixels to scroll content horizontally when user clicks arrow
wPageVer	Word	Number of pixels to scroll content vertically for page
wPageHor	Word	Number of pixels to scroll content horizontally for page
wInfoRefCon	Long	Value passed to draw information bar routine
wInfoHeight	Word	Height of information bar
wFrameDefProc	Long	Pointer to window definition procedure; NIL for standard
wInfoDefProc	Long	Pointer to routine that draws the interior of the information bar
wContDefProc	Long	Pointer to routine that draws the interior of the content region
wPosition	4 words	RECT specifying window's starting position and size
wPlane	Long	Window's starting plane
wStorage	Long	Pointer to memory to use for window record

Each of these parameters is described in more detail in the following paragraphs.

paramLength: Total number of bytes in parameter table, including the *paramLength* parameter itself. Use labels in code to calculate the values, which are used mainly for error checking. Most errors with NewWindow occur because of typing errors occurring when the parameter list is being created. The problem can be compounded if the assembler or compiler skips fields because of the typing errors but does not generate an error.

wFrameBits: Window frame type, as shown in Figure 25-19 (each bit flag is described in more detail after the illustration).

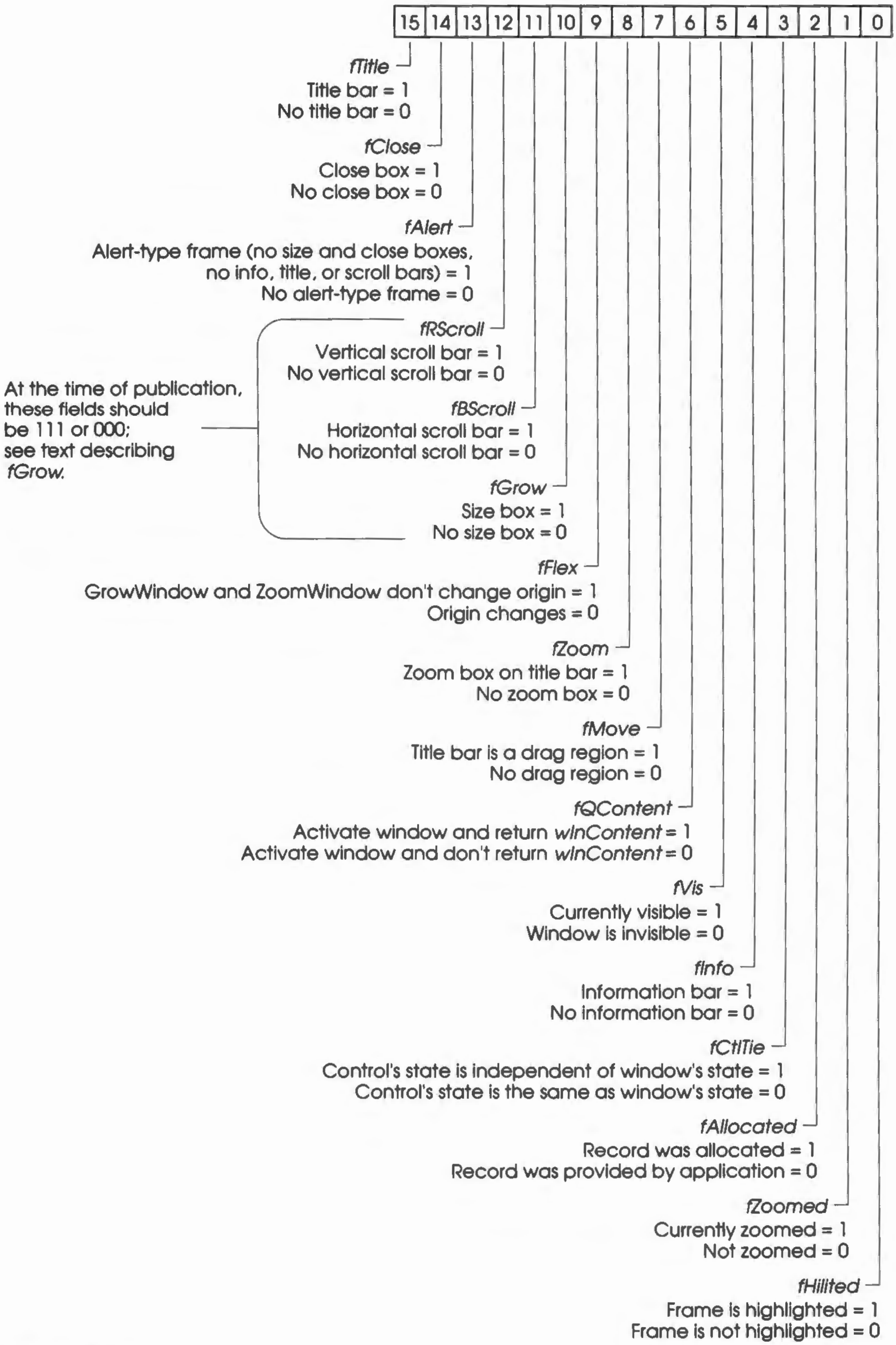


Figure 25-19
NewWindow window frame type

(continued)

fTitle: If this bit is set to 1, the window has a title bar as part of the window frame.

fClose: If this bit is set to 1, the window has a close box as part of the title bar. The window must have a title bar to have a close box.

fAlert: If this flag is set to 1, it indicates to the Dialog Manager that it should draw an alert window. The *fInfo, fZoom, fFlex, fGrow, fBScroll, fRScroll, fClose,* and *fTitle* flags should all be set to 0.

fRScroll: If this bit is set to 1, the window has a right (vertical) scroll bar as part of the window frame.

fBScroll: If this bit is set to 1, the window has a bottom (horizontal) scroll bar as part of the window frame.

fGrow: If this bit is set to 1, the window has a size box as part of the window frame.

❖ *Note:* If *fGrow* is set to 1, *fBScroll* and *fRScroll* must also be set to 1; to have a window frame size box, you must have at least one window frame scroll bar. The fields should be either 111 or 000.

fFlex: If this bit is set to 1, the data height and width are flexible, which means that GrowWindow and ZoomWindow will not change the window's origin as needed.

fZoom: If this bit is set to 1, the window has a zoom box as part of the title bar. The window must have a title bar to have a zoom box.

fMove: If this bit is cleared to 0, the window's title bar is not considered a drag region and therefore the window cannot be moved.

fQContent: If this bit is set to 1 and there is a button-down event inside an inactive window's content, the window will be selected and a wInContent message will be returned by TaskMaster. This feature is useful if you would like to act on any button down in the content, even if it was also used to activate the window.

fVis: If this bit is set to 1, the window is visible.

fInfo: If this bit is set to 1, *wInfoHeight* and *wInfoDefProc* should be given values.

fCtrlTie: When the window is inactive (unhighlighted), its controls are also considered inactive without regard for the active state of the control. Whenever an activate event is received for the window, you should redraw all of the controls for the window to make sure the controls appear in their proper states.

fAllocated: If this flag is set when CloseWindow is called, the window record will be freed. Normally you never have to set or read this flag.

fZoomed: This flag is not used if *fZoom* is 0.

fHilited: This flag will be set by NewWindow, so whatever value you provide will be ignored.

wTitle: Pointer to window's title. If the window has no title bar, this value can be 0. The first byte in the string should be the length of the string followed by the ASCII characters of the title. The title string should always include a space as the first and last character of the string.

wRefCon: Application-defined reference value. This value is reserved for application use and can be any value.

wZoom: RECT data structure specifying size and position of the content region when the window is zoomed. If the bottom side of the rectangle is 0, a default RECT will be used. The default is set so the window uses the entire screen.

wColor: Pointer to window's color table. This is the color table used to draw the window's frame. NIL uses the default color table.

wYOrigin: Vertical offset of content region from data area. This value is the vertical value passed to SetOrigin when TaskMaster is used to draw inside the content region. It is also used to compute the right (or vertical) scroll bar. Set *wYOrigin* to 0 if you are not using window frame scroll bars.

wXOrigin: Horizontal offset of content region from data area. This value is the horizontal value passed to SetOrgin when TaskMaster is used to draw inside the content region. It is also used to compute the bottom (or horizontal) scroll bar. Set *wXOrigin* to 0 if you are not using window frame scroll bars.

wDataH: Height of entire data area. Used to compute the right scroll bar. Set it to 0 if you are not using window frame scroll bars.

wDataW: Width of entire data area. Used to compute the bottom scroll bar. Set it to 0 if you are not using window frame scroll bars.

wMaxH: Maximum content height allowed when growing the window. This value is passed to GrowWindow when called by TaskMaster. If set to 0, a default value will be used so the window will take up the height of the desktop. Set *wMaxH* to 0 if your window frame does not have a size box.

wMaxW: Maximum content width allowed when growing the window. This value is passed to GrowWindow when called by TaskMaster. If set to 0, a default value will be used so the window will take up the width of the desktop. Set *wMaxW* to 0 if your window frame does not have a size box.

wScrollVer: Number of pixels to scroll the content region when the up or down arrows are selected in the right scroll bar. Used only if the scroll bar is part of the frame and TaskMaster is used. Set *wScrollVer* to 0 if you are not using window frame scroll bars.

wScrollHor: Number of pixels to scroll the content region when the left or right arrows are selected in the bottom scroll bar. Used only if the scroll bar is part of the frame and TaskMaster is used. Set *wScrollHor* to 0 if you are not using window frame scroll bars.

(continued)

wPageVer: Number of pixels to scroll the content region when the up or down page regions are selected in the right scroll bar. Used only if the scroll bar is part of the frame and TaskMaster is used. Set *wPageVer* to 0 for the default value of the content region's current height minus 10.

wPageHor: Number of pixels to scroll the content region when the left or right page regions are selected in the bottom scroll bar. Used only if the scroll bar is part of the frame and TaskMaster is used. Set *wPageHor* to 0 for the default value of the content region's current width minus 10.

wInfoRefCon: Value passed to draw information bar routine. The value can be anything the application would like, such as a pointer to a string to be printed in the information bar. Set *wInfoRefCon* to 0 if you are not using an information bar.

wInfoHeight: Height of the information bar if *fInfo* (bit 4) of *wFrame* is set to 1.

wFrameDefProc: Pointer to window's definition procedure; NIL for a standard document window.

wInfoDefProc: Pointer to routine that will be called to draw in the information bar. Set it to 0 if you are not using window frame information bar.

wContDefProc: Pointer to routine that will be called to draw the window's content region. If you are using window frame scroll bars, this value must be set.

If you are not using window frame scroll bars and want to handle update events yourself, set this value to NIL.

If you are not using window frame scroll bars, but you would like TaskMaster to handle update events, set this value. The routine will be called when the content region needs to be drawn. On entry, the current port will be the window's GrafPort, the visible region will be set to the update area, and the origin set. There are no input or output parameters. Exit the routine via RTL.

wPosition: A RECT data structure specifying the window's starting position and size in global coordinates. The RECT becomes the port rectangle of the window's GrafPort; note, however, that the port rectangle is in local coordinates. NewWindow sets the top left corner of the port rectangle to (0,0). For the standard types of windows, this RECT data structure defines the content region of the window.

wPlane: Pointer to window's starting plane; that is, to the window's GrafPort behind which this window should appear—0 for bottommost, $FFFFFFFF for topmost.

wStorage: Pointer to memory to use for window's record. If set to NIL, memory for the record will be allocated by the Window Manager. Because window records are not completely defined, the size needed for one is unknown and you must allow at least 325 bytes for a window record. It is usually best to have the record allocated by the Window Manager. The ability to use your own memory for a window record is provided in case you need to put up a window informing the user that there is no more memory.

$210E PinRect

Pins a specified point inside a specified rectangle. If the point is inside the rectangle, the point is returned; otherwise, the point associated with the nearest pixel within the rectangle is returned. (The high-order word of the pinned point is the X coordinate; the low-order word is the Y coordinate.)

More precisely, for a specified rectangle (*left,top,right,bottom*) and a specified point (*h,v*), PinRect does the following:

- If *h* < *left,* it returns left.
- If *v* < *top,* it returns top.
- If *h* > *right,* it returns right –1.
- If *v* > *bottom,* it returns bottom –1.

❖ *Note:* The 1 is subtracted when the specified point is below or to the right of the specified rectangle so that a pixel drawn at that point will lie within theRect.

Parameters

Stack before call

previous contents	
— *longspace* —	**Long**—Space for result
theXPt	**Word**—X coordinate of point to be pinned
theYPt	**Word**—Y coordinate of point to be pinned
— *theRectPtr* —	**Long**—POINTER to RECT data structure defining boundary of point
	← **SP**

Stack after call

previous contents	
— *pinnedPt* —	**Long**—POINT; high word = X coordinate, low word = Y coordinate
	← **SP**

Errors None

C

```
extern pascal Long PinRect(theXPt,theYPt,theRectPtr)
Integer     theXPt;
Integer     theYPt;
RectPointer     theRectPtr;
```

You can also use the following alternate form of the call:

```
extern pascal Long PinRect(thePoint,theRectPtr)
Point     thePoint;
RectPointer     theRectPtr;
```

❖ *Note:* C Pascal-type functions do not deal properly with data structures returned on the stack. The Long result returned by this call can be passed to any calls requiring a point as a parameter. You cannot use the C dot operator to access the individual Y and X coordinates within the value returned by this call.

$390E RefreshDesktop

Redraws the entire desktop and all the windows. This routine can be useful when the entire screen is clobbered by some application-specific, non–Window Manager operation.

Parameters

Stack before call

previous contents
-- *redrawRect* --
← **SP**

redrawRect: **Long**—POINTER to RECT of rectangle to redraw; NIL for entire screen

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void RefreshDesktop(redrawRect)

Rect *redrawRect;
```

$110E SelectWindow

Makes a specified window the active window. This routine unhighlights the previously active window, brings the specified window in front of all other windows, highlights it, and generates appropriate activate events. Call this routine if you are not using TaskMaster and there's a mouse-down event in the content region of an inactive window.

Parameters

Stack before call

previous contents
-- *theWindowPtr* --   **Long**—POINTER to window's GrafPort
← **SP**

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void SelectWindow(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$140E SendBehind

Changes the position of a specified window, redrawing any exposed windows.

If *behindWindowPtr* is –2 ($FFFFFFFE), it sends the specified window behind all other windows. If *behindWindowPtr* is –1 ($FFFFFFFF), it puts the specified window in front of all other windows. If the specified window is the active window, the routine unhighlights the active window, highlights the new active window, and generates the appropriate activate events.

Parameters

Stack before call

previous contents	
--*behindWindowPtr*--	**Long**—POINTER to GrafPort or $FFFFFFFF = top or $FFFFFFFE = bottom
--*theWindowPtr*--	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SendBehind(behindWindowPtr,theWindowPtr)

GrafPortPtr     behindWindowPtr;

GrafPortPtr     theWindowPtr;
```

$490E SetContentDraw

Sets the pointer to the routine to draw the content region of a specified window.

TaskMaster calls this routine when it gets an update event for that window. See the section "Draw Content Routine" in this chapter for more infomation about the draw routine.

Parameters

Stack before call

previous contents	
— *contentProcPtr* —	**Long**—POINTER to routine to draw content region
— *theWindowPtr* —	**Long**—POINTER to window's GrafPort
← **SP**	

Stack after call

Stack
previous contents
← **SP**

Errors

None

C

```
extern pascal void SetContentDraw(contentDrawPtr,theWindowPtr)

VoidProcPtr      contentDrawPtr;

GrafPortPtr      theWindowPtr;
```

$3F0E SetContentOrigin

Sets the origin of the window's GrafPort when handling an update event. The values are used by TaskMaster to set the origin of the window's GrafPort and are also used by the Window Manager to compute scroll bars in the window frame. See the section "Origin Movement" in this chapter for an illustration of the origin values.

Setting the origin values generates an update event.

Parameters

Stack before call

Stack	
previous contents	
xOrigin	**Word**—Content region's horizontal offset into data area
yOrigin	**Word**—Content region's vertical offset into data area
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetContentOrigin(xOrigin,yOrigin,theWindowPtr)

Word      xOrigin;

Word      yOrigin;

GrafPortPtr     theWindowPtr;
```

$570E SetContentOrigin2

Sets the origin of the window's GrafPort when handling an update event and allows the application to scroll or not scroll the window's content region. If *scrollFlag* is set to 1, this call is the same as SetContentOrigin. If *scrollFlag* is set to 0, the window's origin will be set without the Window Manager scrolling the data in the window. This feature is useful, for example, if a window's data area has to be expanded to the left or above the current origin and you don't want your application to redraw everything in the window.

The *xOrigin and yOrigin* values are used by TaskMaster to set the origin of the window's GrafPort and are also used by the Window Manager to compute scroll bars in the window frame. See the section "Origin Movement" in this chapter for an illustration of the origin values.

Setting the origin values generates an update event.

Parameters

Stack before call

previous contents	
scrollFlag	**Word**—0 to not scroll content, 1 to scroll content
xOrigin	**Word**—Content region's horizontal offset into data area
yOrigin	**Word**—Content region's vertical offset into data area
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetContentOrigin2(scrollFlag,xOrigin,yOrigin,
theWindowPtr)

Word      scrollFlag;
Word      xOrigin;
Word      yOrigin;
GrafPortPtr     theWindowPtr;
```

$410E SetDataSize

Sets the height and width of the data area of a specified window. Setting these values will not change the scroll bars or generate update events.

Parameters

Stack before call

previous contents	
dataWidth	**Word**—Width of data area in pixels
dataHeight	**Word**—Height of data area in pixels
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetDataSize(dataWidth,dataHeight,theWindowPtr)

Word      dataWidth;

Word      dataHeight;

GrafPortPtr      theWindowPtr;
```

$320E SetDefProc

Sets the pointer to the routine that defines a window's frame and behavior. See the section "Defining Your Own Windows" in this chapter for an explanation of what a definition procedure does.

Parameters

Stack before call

previous contents	
-- *wDefProcPtr* --	**Long**—POINTER to window's definition procedure
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetDefProc(wDefProcPtr,theWindowPtr)

LongProcPtr      wDefProcPtr;

GrafPortPtr      theWindowPtr;
```

$0F0E SetFrameColor

Sets the color of a specified window's frame. Does not redraw the window. Do a HideWindow call before the SetFrameColor call and a ShowWindowcall after the SetFrameColor call to redraw the window in its new colors.

The interaction between the *newColorPtr* and *theWindowPtr* parameters is shown in Figure 25-20. See the section "Window Frame Colors and Patterns" in this chapter for a definition of the color table.

Parameters

Stack before call

previous contents	
— *newColorPtr* —	**Long**—POINTER to five-word pattern/color table; NIL for default table
— *theWindowPtr* —	**Long**—POINTER to window's GrafPort; NIL for window default
← **SP**	

Stack after call

Stack
previous contents
← **SP**

Errors

None

C

```
extern pascal void SetFrameColor(newColorPtr,theWindowPtr)

WindColorPtr     newColorPtr;

GrafPortPtr     theWindowPtr;
```

(continued)

The newColorPtr and theWindowPtr parameters

The precise results of the four possible combinations of the *newColorPtr* and *theWindowPtr* parameters are shown in Figure 25-20.

SetFrameColor parameters		
newColorPtr	*theWindowPtr*	**Result**
Pointer	Pointer	Changes a specified window to a specified color table
Pointer	NIL	Makes a specified color table the default table for all future windows
NIL	Pointer	Changes the specified window to the Window Manager's startup color table
NIL	NIL	Replaces the default color table for all future windows with the Window Manager's startup color table

Figure 25-20
SetFrameColor *newColorPtr* and *theWindowPtr* values

$160E SetInfoDraw

Sets the pointer to a routine that draws the information bar for a specified window. If the window has an information bar, the standard window definition procedure calls this routine whenever the window's frame needs to be drawn. See the section "Draw Information Bar Routine" in this chapter for more information about the draw routine.

Parameters

Stack before call

previous contents	
-- *infoDrawPtr* --	**Long**—POINTER to draw information bar routine
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetInfoDraw(infoDrawPtr,theWindowPtr)

VoidProcPtr     infoDrawPtr;

GrafPortPtr     theWindowPtr;
```

$360E SetInfoRefCon

Sets the value associated with the draw information bar routine for a specified window. That value is reserved for use by the applicaton.

Parameters

Stack before call

previous contents	
-- *infoRefCon* --	**Long**—Value to pass to draw information bar routine
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetInfoRefCon(infoRefCon,theWindowPtr)

LongWord     infoRefCon;

GrafPortPtr     theWindowPtr;
```

$430E SetMaxGrow

Sets the maximum values to which a specified window's content region can grow.

Parameters

Stack before call

previous contents	
maxWidth	**Word**—Maximum content width in pixels
maxHeight	**Word**—Maximum content height in pixels
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetMaxGrow(maxWidth,maxHeight,theWindowPtr)

Word      maxWidth;

Word      maxHeight;

GrafPortPtr     theWindowPtr;
```

$340E SetOriginMask

Specifies the mask used to put the horizontal origin on a grid.

SetOriginMask is useful when you are using a scrollable window in 640 mode with dithered colors. In that mode, pixels must keep the same horizontal position to remain the same color. Scrolling windows can change the color by putting the pixels in the wrong horizontal position. SetOriginMask prevents this problem by providing an *originMask* that will be ANDed by TaskMaster with any new horizontal origin to force the origin to certain boundaries. The default is $FFFF, single pixel.

Parameters

Stack before call

Stack	
previous contents	
originMask	**Word**—Mask used to place horizontal origin on a grid
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetOriginMask(originMask,theWindowPtr)

Word      originMask;

GrafPortPtr     theWindowPtr;
```

$470E SetPage

Sets the number of pixels by which TaskMaster will scroll the content region when the user selects the page regions on window frame scroll bars.

Parameters

Stack before call

previous contents	
hPage	**Word**—Number of pixels to page horizontally
vPage	**Word**—Number of pixels to page vertically
— *theWindowPtr* —	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetPage(hPage,vPage,theWindowPtr)

Word      hPage;

Word      vPage;

GrafPortPtr     theWindowPtr;
```

$450E SetScroll

Sets the number of pixels by which TaskMaster will scroll the content region when the user selects the arrows on window frame scroll bars.

Parameters

Stack before call

previous contents	
hScroll	**Word**—Number of pixels to scroll horizontally
vScroll	**Word**—Number of pixels to scroll vertically
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetScroll(hScroll,vScroll,theWindowPtr)

Word      hScroll;

Word      vScroll;

GrafPortPtr     theWindowPtr;
```

$4B0E SetSysWindow

Marks a specified window as a system window.

Parameters

Stack before call

previous contents
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
← **SP**

Stack after call

previous contents
← **SP**

Errors

None

C

```
extern pascal void SetSysWindow(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$2D0E SetWFrame

Sets the bit flag that describes a specified window's frame type. The window frame is not redrawn. See the discussion of *wFrameBits* in the section "NewWindow" in this chapter for the definition of the bits of the *wFrame* parameter.

❖ *Note:* Normally, you won't need to call this routine; instead, you should set up the window frame correctly with the NewWindow routine. The SetWFrame routine is provided for custom window definition procedures.

Parameters

Stack before call

Stack	
previous contents	
wFrame	**Word**—Bit flag specifying window's frame type
— *theWindowPtr* —	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetWFrame(wFrame,theWindowPtr)

Word        wFrame;

GrafPortPtr     theWindowPtr;
```

$4E0E SetWindowIcons

Sets the icon font for the Window Manager. See the section "Window Manager Icon Font" in this chapter for more information about the font.

If you want to use the call to simply retrieve the handle of the current font, specify a negative value for *newFontHandle*.

Parameters

Stack before call

previous contents	
longspace	**Long**—Space for result
newFontHandle	**Long**—HANDLE to new icon font; negative to not replace font
← **SP**	

Stack after call

previous contents	
oldFontHandle	**Long**—HANDLE to icon font before call
← **SP**	

Errors

None

C

```
extern pascal FontHndl SetWindowIcons(newFontHandle)

FontHndl     newFontHandle;
```

$280E SetWRefCon

Sets a value that is inside a specified window record and is reserved for the application's use.

Parameters

Stack before call

previous contents	
-- *wRefCon* --	**Long**—Value of *wRefCon* field
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetWRefCon(wRefCon,theWindowPtr)

Longint      wRefCon;

GrafPortPtr     theWindowPtr;
```

$0D0E SetWTitle

Changes the title of a specified window to a specifed title and redraws the window. The string pointed to by *titlePtr* must be a Pascal-type string.

Warning
The string pointed to by *titlePtr* must not be changed or moved.

Parameters

Stack before call

previous contents	
-- *titlePtr* --	**Long**—POINTER to new title
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetWTitle(titlePtr,theWindowPtr)

Pointer      titlePtr;

GrafPortPtr      theWindowPtr;
```

$380E SetZoomRect

Sets the rectangle to be used as the content's zoomed or unzoomed size for a specified window. If the window is currently in a zoomed state (that is, bit 2, the *fZoomed* bit, is set to 1 in the *wFrame* flag—see the section "NewWindow" in this chapter), *wZoomSizePtr* should point to a RECT data structure that specifies the window's unzoomed size and position.

If the window is currently in an unzoomed state (that is, the *fZoomed* bit is set to 0), *wZoomSizePtr* should point to a RECT data structure that specifies the zoomed size and position of the window. The rectangle will be used as the window's content region (port) the next time the window is zoomed by a call to ZoomWindow.

Parameters

Stack before call

previous contents	
-- *wZoomSizePtr* --	**Long**—POINTER to rectangle to be used as content's zoomed size
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SetZoomRect(wZoomSizePtr,theWindowPtr)
Rect *wZoomSizePtr;
GrafPortPtr     theWindowPtr;
```

$230E ShowHide

Shows or hides a window. If *showFlag* is TRUE, ShowHide makes the specified window visible if it's not already visible and has no effect if it is already visible. If *showFlag* is FALSE, ShowHide makes the window invisible if it's not already invisible and has no effect if it is already invisible.

Unlike HideWindow and ShowWindow, ShowHide never generates activate events or changes the highlighting or front-to-back ordering of windows.

Important

Use this procedure carefully and only in special circumstances in which you need more control than is allowed by ShowWindow and HideWindow. You could end up with an active window that isn't highlighted.

Parameters

Stack before call

previous contents	
showFlag	**Word**—BOOLEAN; TRUE to show, FALSE to hide
— *theWindowPtr* —	**Long**—POINTER to window's GrafPort
← **SP**	

Stack after call

Stack
previous contents
← **SP**

Errors

None

C

```
extern pascal void ShowHide(showFlag,theWindowPtr)

Boolean      showFlag;

GrafPortPtr     theWindowPtr;
```

$130E ShowWindow

Makes a specified window visible if it was invisible and then draws the window. It does not change the front-to-back ordering of the windows. If you have previously hidden the frontmost window with HideWindow, HideWindow will have brought the window behind it to the front. If you then do a ShowWindow of the window you hid, it will no longer be frontmost.

If the specified window is already visible, ShowWindow has no effect.

Parameters

Stack before call

previous contents	
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void ShowWindow(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$1C0E SizeWindow

Enlarges or shrinks the port rectangle of the specified window's GrafPort to a specified width and height. If the new width and height are specified as 0, SizeWindow does nothing. The window's position on the screen does not change. When the new window frame is drawn, if the width of a document window changes, the title is recentered in the title bar, or if it no longer fits, it is truncated.

Parameters

Stack before call

previous contents	
newWidth	**Word**—New width of window in pixels
newHeight	**Word**—New height of window in pixels
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void SizeWindow(newWidth,newHeight,theWindowPtr)

Word      newWidth;

Word      newHeight;

GrafPortPtr     theWindowPtr;
```

$4D0E StartDrawing

Makes a specified window the current port and sets its origin. After the call, any drawing occurs inside the specified window's content area and in the proper coordinate system.

Important

Do not call StartDrawing between a BeginUpdate and an EndUpdate call. Also, when you have finished drawing, call the QuickDraw II routine SetOrigin(0,0).

StartDrawing can be used for drawing in a window's content region outside of update events.

❖ *Note:* StartDrawing is useful only with standard document windows with frame scroll bars. Otherwise, only a SetPort call is needed to make the correct port current.

Parameters

Stack before call

previous contents	
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void StartDrawing(theWindowPtr)

GrafPortPtr     theWindowPtr;
```

$500E StartInfoDrawing

Allows an application to draw or hit test outside of its information bar definition procedure. If there is no information in the window, the coordinates of the RECT data structure pointed to by *infoRectPtr* will all be 0. The coordinate system will be local to the window's frame; that is, 0,0 will be the upper left corner of the window, and the current GrafPort will be the Window Manager's.

You can set the clip region after a StartInfoDrawing call and before an EndInfoDrawing call; this is not true from within the information bar definition procedure.

Important

When you finish dealing with the information bar, you must call the EndInfoDrawing routine before you make any other calls to the Window Manager.

Parameters

Stack before call

Stack	
previous contents	
— *infoRectPtr* —	**Long**—POINTER to RECT where information bar's data will be stored
— *theWindowPtr* —	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

Stack	
previous contents	
	← **SP**

Errors

None

C

```
extern pascal void StartInfoDrawing(infoRectPtr,theWindowPtr)

Rect *infoRectPtr;

GrafPortPtr     theWindowPtr;
```

$1D0E TaskMaster

Calls GetNextEvent and looks in the event part of the task record to see if it can handle the event. If no event is returned by GetNextEvent, 0 is returned in *taskCode*. For further description of TaskMaster's activities, see the following section "TaskMaster Pseudocode."

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
taskMask	**Word**—Mask used by TaskMaster to call GetNextEvent
— *taskRecPtr* —	**Long**—POINTER to Window Manager task record
	← **SP**

Stack after call

previous contents	
taskCode	**Word**—Code indicating action to be taken; 0 = no further task to perform
	← **SP**

Errors

$0E03	`taskMaskErr`	Reserved bits not clear in *wmTaskMask* field of task record

C

```
extern pascal Word TaskMaster(taskMask,taskRecPtr)

Word      taskMask;

WmTaskRecPtr     taskRecPtr;
```

TaskMaster pseudocode

TaskMaster can be a powerful tool, and can take care of much of the mundane processing of events. Because the routine is so important, this section presents pseudocode describing TaskMaster's operations.

To perform its functions, TaskMaster takes the following steps:

Calls the Desk Manager routine SystemTask.

Calls the Event Manager routine GetNextEvent with a TaskRec and *eventMask*.

The *wmMessage* field of TaskRec is duplicated into the *wmTaskData* field of TaskRec.

If any of the reserved bits in the *wmTaskMask* field are not 0:

{

Low word of *wmTaskData* = 0.
Returns `nullEvt` ($0000).
Error returned: `wmTaskMaskErr` ($0E03).

}

If *wmWhat* field of TaskRec = `nullEvt` ($0000):

{

Low word of TaskData = 0.
Returns `nullEvt` ($0000).

}

If *wmWhat* field of TaskRec = `updateEvt` ($0006):

{

If *wmTaskMask* bit *tmUpdate* (bit 1) = 0:

{

wmTaskData = pointer to window to be updated.
Returns `updateEvt` ($0006).

}

If window's *wContDefProc* field = 0:

{

wmTaskData = pointer to window to be updated.
Returns `updateEvt` ($0006).

}

Calls the BeginUpdate routine.

The window's draw routine in window's *wContDefProc* field is called (routine in application).

Calls the EndUpdate routine.

wmTaskData low word = `updateEvt` ($0006).
Returns `nullEvt` ($0000).

}

(continued)

If *wmWhat* field of TaskRec = `activateEvt` ($0008):

{

  If *wmTaskMask* bit *tmCRedraw* (bit 13) = 1:

  {

    If *wframe* bit *fCtlTie* (bit 3) = 1:

    {

      Calls the Control Manager routine DrawControls to draw controls in proper state.

    }

  }

  wmTaskData = pointer to window that was activated or deactivated (check modifier field).

  Returns `activateEvt` ($0008).

}

If *wmWhat* field of TaskRec = `keyDownEvt` ($0003) OR `autoKeyEvt` ($0005):

{

  If *wmTaskMask* bit *tmMenuKey* (bit 0) = 0:

  {

    wmTaskData = message field as returned by GetNextEvent.

    Returns `keyDownEvt` ($0003).

  }

  Calls the Menu Manager routine MenuKey with the given TaskRec for the system menu bar.

  Go to Menu Selection.

  (The remainder of TaskMaster from this point is the same as when the Menu Manager routine MenuSelect is called.)

}

If *wmWhat* field of TaskRec does not equal `mouseDownEvt` ($0001):

{

  Returns *what* field from TaskRec.

}

```
If wmWhat field of TaskRec = mouseDownEvt($0001):
{
        If TaskMask bit tmFindW (bit 2) = 0:
        {
                wmTaskData = message field from GetNextEvent.
                Returns mouseDownEvt ($0001).
        }

        Calls FindWindow.

        If FindWindow returns wInMenuBar ($0011):
        {
                If TaskMask tmMenuSel (bit 3) = 0:
                {
                        Low word of wmTaskData = 0.
                        Returns wInMenuBar ($0011).
                }

                MenuSelect is called with TaskRec passed to TaskMaster.

Menu Selection:
                If low word of wmTaskData = 0, then no selection made:
                {
                        If wmTaskMask bit tmInactive (bit 14) = 0:
                        {
                                Low word of wmTaskData = wInMenuBar ($0011).
                                Returns nullEvt ($0000).
                        }

                        If high word of wmTaskData = nonzero:
                        {
                                Low word of wmTaskData = 0.
                                High word of wmTaskData = ID of selected inactive
                                menu item.
                                Returns wInactMenu ($001C).
                        }
                        Low word of wmTaskData = wInMenuBar ($0011).
                        Returns nullEvt ($0000).
                }
```

(continued)

Else, menu selection made:
{
If low word of *wmTaskData* (menu item ID) < 250:
{
If *wmTaskMask* bit *tmOpenNDA* (bit 4) = 0:
{
Low word of *wmTaskData* = ID of selected menu item.
High word of *wmTaskData* = ID of menu from which selection was made.
Returns `wInDeskItem` ($001A).
}

Calls the Desk Manager routine OpenNDA to open the desk accessory selected.
Calls Menu Manager routine HiliteMenu to unhighlight the selected menu.
Low word of *wmTaskData* = `wInDeskItem` ($001A).
Returns `nullEvt` ($0000).
}
If TaskMask bit *tmSpecial* (bit 12) = 0:
{
Low word of *wmTaskData* = ID of selected menu item.
High word of *wmTaskData* = ID of menu from which selection was made.
Returns `wInSpecial` ($0019).
}
If top window is an application (nonsystem) window:
{
Low word of *wmTaskData* = ID of selected menu item.
High word of *wmTaskData* = ID of menu from which selection was made.
Returns `wInSpecial` ($0019).
}

If low word of *wmTaskData* (menu item ID) = 255 (Close item):
{
Calls Desk Manager routine CloseNDAbyWinPtr for top window (system window).
Calls HiliteMenu to unhighlight the selected menu.
Low word of *wmTaskData* = `wClosedNDA` ($001D).
Returns `nullEvt` ($0000).
}

If low word of *wmTaskData* (menu item ID) = 250, 251, 252, 253, or 254 (edit item):
{
Calls Desk Manager routine SystemEdit with ID of special edit menu item.
If SystemEdit returns FALSE:
{
Low word of *wmTaskData* = ID of selected menu item.
High word of *wmTaskData* = ID of menu from which selection was made.
Returns `wInSpecial` ($0019).
}
(Top system window handled the special menu item selection.)
Calls the Menu Manager routine HiliteMenu to unhighlight the selected menu.
Low word of TaskData = `wCalledSysEdit` ($001E).
Returns `nullEvt` ($0000).
}
(Low word of *wmTaskData* (menu item ID) > 255.)
Low word of *wmTaskData* = ID of selected menu item.
High word of *wmTaskData* = ID of menu from which selection was made.
Returns `wInMenuBar` ($0011).
} (end menu selection)
} (end FindWindow `wInMenuBar`)

If FindWindow returns a negative value:
{
If *wmTaskMask* bit *tmSysClick* (bit 5) = 0:
{
wmTaskData = window pointer returned from FindWindow.
Returns result from FindWindow.
}
Calls Desk Manager routine SystemClick with result from FindWindow.
wmTaskData low word = `wClickCalled` ($0012).
Returns `nullEvt` ($0000).
}

(continued)

If FindWindow returns `wInDrag` ($0014):

{

If *wmTaskMask* bit *tmDragW* (bit 6) = 0:

{

wmTaskData = window pointer returned from FindWindow.

Returns `wInDrag` ($0014).

}

If bit 8 in the modifier field of TaskRec (Apple key up) and the window is not active:

{

Calls SelectWindow to make window active.

}

Calls DragWindow.

wmTaskData = `wInDrag` ($0014).

Returns `nullEvt` ($0000).

}

If FindWindow returns `wInContent` ($0013):

{

If *wmTaskMask* bit *tmContent* (bit 7) = 0:

{

wmTaskData = window pointer returned from FindWindow.

Returns `wInContent` ($0013).

}

If the window is not active:

{

Calls SelectWindow to make window active.

}

If *wFrame* field *fQContent* (bit 6) = 1:

{

wmTaskData = window pointer returned from FindWindow.

Returns `wInContent` ($0013).

}

Low word of *wmTaskData* = `wInContent` ($0013).

Returns `nullEvt` ($0000).

}

```
If FindWindow returns wInGoAway ($0016):
{
        If wmTaskMask bit tmClose (bit 8) = 0:
        {
                wmTaskData = window pointer returned from FindWindow.
                Returns wInGoAway ($0016).
        }
        Calls TrackGoAway.
        If TrackGoAway returns TRUE:
        {
                wmTaskData = window pointer returned from FindWindow.
                Returns wInGoAway ($0016).
        }
        Low word of wmTaskData = wInGoAway ($0016).
        Returns nullEvt ($0000).
}
If FindWindow returns wInZoom ($0017):
{
        If wmTaskMask bit tmZoom (bit 9) = 0:
        {
                wmTaskData = window pointer returned from FindWindow.
                Returns wInZoom ($0017).
        }
        Calls TrackZoom.
        If TrackZoom returns TRUE:
        {
                Calls ZoomWindow.
                Low word of wmTaskData = wInZoom ($0017).
                Returns nullEvt ($0000).
        }
        Low word of wmTaskData = wTrackZoom ($001F).
        Returns nullEvt ($0000).
}
If FindWindow returns wInGrow ($0015):
{
        If wmTaskMask bit tmGrow (bit 10) = 0:
        {
                wmTaskData = window pointer returned from FindWindow.
                Returns wInGrow ($0015).
        }
        Calls GrowWindow.
        Calls SizeWindow with results from GrowWindow.
        Low word of wmTaskData = wInGrow ($0015).
        Returns nullEvt ($0000).
}
```

(continued)

```
If FindWindow returns wInInfo ($0018):
{
        If wmTaskMask bit tmInfo (bit 15) = 1:
        {
                If window is not active:
                {
                        Calls SelectWindow.
                        Low word of wmTaskData = wInInfo ($0018).
                        Returns nullEvt ($0000).
                }
        }
        wmTaskData = window pointer returned from FindWindow.
        Returns wInInfo ($0018).
}

If FindWindow returns wInFrame ($001B):
{
        If wmTaskMask bit tmScroll (bit 11) = 0:
        {
                wmTaskData = window pointer returned from FindWindow.
                Returns wInFrame ($001B).
        }
        If window is not active:
        {
                Calls SelectWindow to make active.
                Low word of wmTaskData = wHitFrame ($0020).
                Returns nullEvt ($0000).
        }
        If button was on a window frame control:
        {
                Low word of wmTaskData = wHitFrame ($0020).
                Returns nullEvt ($0000).
        }
        Calls TrackControl with an action procedure within TaskMaster.
        The action procedure in TrackMaster performs scrolling and
        updates.
        Low word of wmTaskData = wInFrame ($001B).
        Returns nullEvt ($0000).
}

Else (something returned from FindWindow other than those handled
above):
{
        wmTaskData = returned value from FindWindow.
        Returns result from FindWindow.
}
```

$180E TrackGoAway

Tracks the mouse until the mouse button is released, highlighting the go-away region as long as the mouse location remains inside it and unhighlighting it when the mouse moves outside it.

When there's a mouse-down event in the go-away region of the specified window and the application is not using TaskMaster, the application should call TrackGoAway with *startX* and *startY* equal to the point where the mouse button was pressed (in global coordinates, as stored in the *where* field of the event record).

The exact way a window's go-away region is highlighted depends on its window definition procedure. If the user releases the mouse button while the cursor is inside the go-away region, TrackGoAway unhighlights the go-away region and returns TRUE (following which the application should eventually perform a CloseWindow). If the user releases the mouse button while the cursor is outside the go-away region, TrackGoAway returns FALSE (in which case the application should do nothing).

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
startX	**Word**—Starting X coordinate of cursor, in global coordinates
startY	**Word**—Starting Y coordinate of cursor, in global coordinates
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
goAway	**Word**—BOOLEAN; TRUE if go-away selected when button released, FALSE if not
	← **SP**

Errors

None

C

```
extern pascal Boolean TrackGoAway(startX,startY,theWindowPtr)
Integer     startX;
Integer     startY;
GrafPortPtr     theWindowPtr;
```

You can also use the following alternate form of the call:

```
extern pascal Boolean TrackGoAway(start,theWindowPtr)
Point     start;
GrafPortPtr     theWindowPtr;
```

$260E TrackZoom

Tracks the mouse until the mouse button is released, highlighting the zoom region as long as the mouse location remains inside it and unhighlighting it when the mouse moves outside it.

When there's a mouse-down event in the zoom region of the specified window and the application is not using TaskMaster, the application should call TrackZoom with *startX* and *startY* equal to the point where the mouse button was pressed (in global coordinates, as stored in the *where* field of the event record).

The exact way a window's zoom region is highlighted depends on its window definition procedure. If the mouse button is released inside the zoom region, TrackZoom unhighlights the zoom region and returns TRUE (following which the application should eventually perform a ZoomWindow). If the mouse button is released outside the zoom region, TrackZoom returns FALSE (in which case the application should do nothing).

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
startX	**Word**—Starting X coordinate of cursor, in global coordinates
startY	**Word**—Starting Y coordinate of cursor, in global coordinates
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
zoom	**Word**—BOOLEAN; TRUE if zoom region was selected, FALSE if not
	← **SP**

Errors None

C

```
extern pascal Boolean TrackZoom(startX,startY,theWindowPtr)
Integer     startX;
Integer     startY;
GrafPortPtr     theWindowPtr;
```

You can also use the following alternate form of the call:

```
extern pascal Boolean TrackZoom(start,theWindowPtr)
Point     start;
GrafPortPtr     theWindowPtr;
```

$3C0E ValidRect

Removes a specified rectangle from the update region of the window whose GrafPort is the current port and tells the Window Manger to cancel any updates accumulated for that rectangle. The rectangle is clipped to the window's content region and is given in local coordinates.

Important

This routine changes the coordinates you give it. Save the coordinates if you need to restore them later.

Parameters

Stack before call

previous contents	
-- *goodRectPtr* --	**Long**—POINTER to RECT specifying rectangle to be removed
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void ValidRect(goodRectPtr)

Rect *goodRectPtr;
```

$3D0E ValidRgn

Removes a specified region from the update region of the window whose GrafPort is the current port and tells the Window Manger to cancel any updates accumulated for that region. The region is clipped to the window's content region and is given in local coordinates.

Important

This routine changes the coordinates you give it. Save the coordinates if you need to restore them later.

Parameters

Stack before call

previous contents	
-- *goodRgnHandle* --	**Long**—HANDLE to region to be subtracted from update region
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void ValidRgn(goodRgnHandle)

RgnHandle     goodRgnHandle;
```

$530E WindDragRect

Pulls a dotted outline of a specified rectangle around the screen, following the movements of the mouse until the mouse button is released. WindDragRect is a way of accessing the Control Manager DragRect routine. See the section "DragRect" in Chapter 4, "Control Manager," in Volume 1 for more information.

Parameters

Stack before call

previous contents	
— *longspace* —	**Long**—Space for result
— *actionProcPtr* —	**Long**—POINTER to routine; NIL for default
— *dragPatternPtr* —	**Long**—POINTER to pattern to use for drag outline
startX	**Word**—X coordinate of starting point, in global coordinates
startY	**Word**—Y coordinate of starting point, in global coordinates
— *dragRectPtr* —	**Long**—POINTER to RECT data structure of rectangle to be dragged
— *limitRectPtr* —	**Long**—POINTER to RECT of limit rectangle
— *slopRectPtr* —	**Long**—POINTER to RECT of slop rectangle
dragFlag	**Word**—Bit flag customizing drag rectangle (see Figure 4-25 in Volume 1)
	← **SP**

Stack after call

previous contents	
— *moveDelta* —	**Long**—High word = amount X changed; low word = amount Y changed
	← **SP**

Errors None

C

```
extern pascal LongWord WindDragRect(actionProcPtr,dragPatternPtr,
startX,startY,dragRectPtr,limitRectPtr,slopRectPtr,dragFlag)

VoidProcPtr     actionProcPtr;

Pattern     dragPatternPtr;

Integer     startX;

Integer     startY;

Rect *dragRectPtr;

Rect *limitRectPtr;

Rect *slopRectPtr;

Word     dragFlag;
```

You can also use the following alternate form of the call:

```
extern pascal LongWord WindDragRect(actionProcPtr,dragPatternPtr,
start,dragRectPtr,limitRectPtr,slopRectPtr,dragFlag)

VoidProcPtr     actionProcPtr;

Pattern     dragPatternPtr;

Point     start;

Rect *dragRectPtr;

Rect *limitRectPtr;

Rect *slopRectPtr;

Word     dragFlag;
```

$250E WindNewRes

Closes the Window Manager's GrafPort and opens a new GrafPort in the other Super Hi-Res resolution. However, the screen is not redrawn by the Window Manager in the new resolution. You can then call the RefreshDesktop routine when all resolution changes, such as changes to the desktop pattern and window colors, have been completed.

You should call WindNewRes after the screen resolution has been changed.

Parameters

The stack is not affected by this call. There are no input or output parameters.

Errors

None

C

```
extern pascal void WindNewRes()
```

$560E WindowGlobal

Specifies a mask that determines how the Window Manager performs tasks. If *windowGlobalMask* has bit 15 set to 1, the mask will be ANDed with the global flag. If *windowGlobalMask* has bit 15 set to 0, the mask will be ORed with the global flag. WindowGlobal also returns the current state of the global flag in *windowGlobalFlag*.

Important
At the time of publication, all bits except 15 and 0 are reserved for future use.

Parameters

Stack before call

previous contents	
wordspace	**Word**—Space for result
windowGlobalMask	**Word**—Global mask (see Table 25-11)
	← **SP**

Stack after call

previous contents	
windowGlobalFlag	**Word**—Current state of window global flag (see Table 25-11)
	← **SP**

Errors

None

C

```
extern pascal Word WindowGlobal(windowGlobalMask)

Word     windowGlobalMask;
```

Stopping window highlighting

The only valid values for the window global mask at the time of publication are used to change the Window Manager's normal highlighting procedure, as shown in Table 25-10.

Table 25-10
Window global mask values

Value	Description
$0000	Flag does not change (used to retrieve current state of flag)
$0001	Stop the Window Manager from highlighting and unhighlighting windows when NewWindow and CloseWindow calls are made
$FFFE	Return the Window Manager to normal highlighting operation

Under normal circumstances, the Window Manager highlights and unhighlights windows appropriately when they become active or inactive. However, you may wish to suppress that highlighting in order to speed up window redrawing. For example, the following pseudocode sequence demonstrates one use of WindowGlobal:

```
NewWindow(appropriate parameters) ; Put up a new window, which is
                                  ; automatically highlighted.
WindowGlobal($0001)               ; Turn highlighting off.
doAlert()                         ; Put up an alert window, allow the user
                                  ; to choose something, and close the
                                  ; alert.  Although the alert window is
                                  ; on top of the document window, the
                                  ; document window remains highlighted.
WindowGlobal($FFFE)               ; Return the Window Manager to normal
                                  ; highlighting operations.
```

The value returned in *windowGlobalFlag* indicates the state of the flag after any changes have been made. The values at the time of publication are shown in Table 25-11.

Table 25-11
Window global flag values

Value	Description
$0000	Normal highlighting off
$0001	Normal highlighting on

$270E ZoomWindow

Switches the size and position of a specified window between its current size and position and its maximum size. If the routine is called again before the specified window is moved or resized, the window will be resized and positioned to the size and position before the last ZoomWindow was performed. When a SizeWindow or MoveWindow is performed while a window is zoomed, the last size becomes the new size and position.

Parameters

Stack before call

previous contents	
-- *theWindowPtr* --	**Long**—POINTER to window's GrafPort
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors

None

C

```
extern pascal void ZoomWindow(theWindowPtr)

GrafPortPtr    theWindowPtr;
```

Window Manager summary

This section briefly summarizes the constants, data structures, and tool set error codes contained in the Window Manager.

Important

These definitions are provided in the appropriate interface file.

Table 25-12
Window Manager constants

Name	Value	Description
Axis parameters		
wNoConstraint	$0000	No constraint on movement
wHAxisOnly	$0001	Horizontal axis only
wVAxisOnly	$0002	Vertical axis only
Desktop commands		
FromDesk	$00	Subtract region from desktop
ToDesk	$01	Add region to desktop
GetDesktop	$02	Get handle to desktop region
SetDesktop	$03	Set handle to desktop region
GetDeskPat	$04	Address of pattern or drawing routine
SetDeskPat	$05	Change address of pattern or drawing routine
GetVisDesktop	$06	Get desktop region minus visible windows
BackGroundRgn	$07	For drawing directly on desktop
SendBehind values		
toBottom	–2	Send window to bottom
topMost	–1	Make window frontmost
bottomMost	$0000	Make window bottom

(continued)

Table 25-12 (continued)
Window Manager constants

Name	Value	Description
Task mask values		
`tmMenuKey`	$0001	Handle menu key events
`tmUpdate`	$0002	Handle update events
`tmFindW`	$0004	FindWindow called
`tmMenuSel`	$0008	MenuSelect called
`tmOpenNDA`	$0010	OpenNDA called
`tmSysClick`	$0020	SystemClick called
`tmDragW`	$0040	DragWindow called
`tmContent`	$0080	Activate inactive window on click in content region
`tmClose`	$0100	TrackGoAway called
`tmZoom`	$0200	TrackZoom called
`tmGrow`	$0400	GrowWindow called
`tmScroll`	$0800	Enable scrolling and activate inactive window on click in scroll bar
`tmSpecial`	$1000	Handle events in special menu items
`tmCRedraw`	$2000	Redraw controls upon activate event
`tmInactive`	$4000	Allows selection of inactive menu items
`tmInfo`	$8000	Don't activate inactive window on click in information bar
TaskMaster codes		
`wInDesk`	$0010	In Desktop
`wInMenuBar`	$0011	In system menu bar
`wClickCalled`	$0012	System click called
`wInContent`	$0013	In content region
`wInDrag`	$0014	In drag region
`wInGrow`	$0015	In grow region, active window only
`wInGoAway`	$0016	In go-away region, active window only
`wInZoom`	$0017	In zoom region, active window only
`wInInfo`	$0018	In information bar
`wInSpecial`	$0019	Item ID selected was 250–255
`wInDeskItem`	$001A	Item ID selected was 1–249
`wInFrame`	$001B	In Frame, but not on anything else
`wInactMenu`	$001C	Inactive menu item selected
`wClosedNDA`	$001D	Desk accessory closed
`wCalledSysEdit`	$001E	Inactive menu item selected
`wInSysWindow`	$8000	High-order bit set for system windows

Table 25-12 (continued)
Window Manager constants

Name	Value	Description
varCode values		
`wDraw`	$00	Draw window frame command
`wHit`	$01	Hit test command
`wCalcRgns`	$02	Compute regions command
`wNew`	$03	Initialization command
`wDispose`	$04	Dispose command
wFrame values		
`fHilited`	$0001	Window is highlighted
`fZoomed`	$0002	Window is zoomed
`fAllocated`	$0004	Window record was allocated
`fCtlTie`	$0008	State of window's controls tied to state of window
`fInfo`	$0010	Window has an information bar
`fVis`	$0020	Window is visible
`fQContent`	$0040	Select window if `mouseDownEvt` in inactive window's content
`fMove`	$0080	Window can be dragged
`fZoom`	$0100	Window has a zoom box
`fFlex`	$0200	Data height and width are flexible
`fGrow`	$0400	Window has a size box
`fBScroll`	$0800	Window has a horizontal scroll bar
`fRScroll`	$1000	Window has a vertical scroll bar
`fAlert`	$2000	Alert-type window frame
`fClose`	$4000	Window has a close box
`fTitle`	$8000	Window has a title bar
Record sizes		
`windSize`	$145	Size of window record
`wmTaskRecSize`	$16	Size of task record

Table 25-13
Window Manager data structures

Name	Offset	Type	Definition
WindRec (window record)			
wNext	$00	GrafPortPtr	Pointer to next window record
port	$04	GrafPort	Window's port
wPadding	$AE	16 bytes	Space for possible future expansion
wStrucRgn	$BE	RgnHandle	Region of frame plus content
wContRgn	$C2	RgnHandle	Content region
wUpdateRgn	$C6	RgnHandle	Update region
wControls	$CA	CtlRecHndl	Window's control list
wFrameCtrls	$CE	CtlRecHndl	Window frame's control list
wFrame	$D2	Word	Bit flags
WindColor (window color table)			
frameColor	$00	Word	Color of window frame
titleColor	$02	Word	Color of title and bar
tBarColor	$04	Word	Color and pattern of title bar
growColor	$06	Word	Color of grow box
infoColor	$08	Word	Color of information bar
Paramlist (NewWindow parameter list)			
paramLength	$00	Word	Total number of bytes in parameter table
wFrameBits	$02	Word	Bit flag that describes window
wTitle	$04	Pointer	Pointer to window's title
wRefCon	$08	LongWord	Reserved for application use
wZoom	$0C	Rect	Size and positon of content
wColor	$14	WindColorPtr	Pointer to window's color table
wYOrigin	$18	Integer	Content's vertical origin
wXOrigin	$1A	Integer	Content's horizontal origin
wDataH	$1C	Word	Height of entire document
wDataW	$1E	Word	Width of entire document
wMaxH	$20	Word	Maximum height of content allowed by GrowWindow
wMaxW	$22	Word	Maximum width of content allowed by GrowWindow

Table 25-13 (continued)
Window Manager data structures

Name	Offset	Type	Definition
Paramlist (NewWindow parameter list)			
wScrollVer	$24	Word	Number of pixels to scroll vertically when arrow is clicked
wScrollHor	$26	Word	Number of pixels to scroll horizontally when arrow is clicked
wPageVer	$28	Word	Number of pixels to scroll vertically for page
wPageHor	$2A	Word	Number of pixels to scroll horizontally for page
wInfoRefCon	$2C	LongWord	Value passed to draw information bar routine
wInfoHeight	$30	Word	Height of information bar
wFrameDefProc	$32	LongProcPtr	Pointer to standard window definition procedure
wInfoDefProc	$36	VoidProcPtr	Pointer to routine that draws the interior of the information bar
wContDefProc	$3A	VoidProcPtr	Pointer to routine that draws the interior of the content region
wPosition	$3E	Rect	Window's starting position and size
wPlane	$46	GrafPortPtr	Window's starting plane
wStorage	$4A	WindRecPtr	Pointer to memory to use for window record
WmTaskRec (task record)			
wmWhat	$00	Word	Unchanged from event record
wmMessage	$02	LongWord	Unchanged from event record
wmWhen	$06	LongWord	Unchanged from event record
wmWhere	$0A	Point	Unchanged from event record
wmModifiers	$0E	Word	Unchanged from event record
wmTaskData	$10	LongWord	TaskMaster return value
wmTaskMask	$14	LongWord	TaskMaster feature mask

Note: The actual assembly-language equates have a lowercase letter *o* in front of all the names given in this table.

Table 25-14
Window Manager error codes

Code	Name	Description
$0E01	`paramLenErr`	First word of parameter list is the wrong size
$0E02	`allocateErr`	Unable to allocate memory for window record
$0E03	`taskMaskErr`	Reserved bits not clear in *wmTaskMask* field of task record

Appendix A

Writing Your Own Tool Set

The Tool Locator system, which is flexible enough to allow you to write your own tool sets for use in your applications, supports both system tools and user tools.

When writing your own tool set, the following must be kept in mind:

- Tool sets get control in full native mode.
- Work space should be dynamically assigned. Tool sets should not use any fixed RAM locations for work space; they should obtain their work space from the Memory Manager. This avoids memory conflicts, such as those caused by fixed usage of screen holes.
- A simple interrupt environment should be supplied. Each function should increment or decrement the busy flag, be reentrant, or disable interrupts during execution. The most common approach is to use the busy flag. See Chapter 19, "Scheduler," for more information.
- Before returning control to the caller, routines must restore the caller's execution environment. This includes the data bank register, the direct-page register, and any soft switches.
- Routines should not assume the presence of any operating system unless the operating system is directly relevant; for example, a routine that reads or writes a file, where other considerations demand that the file type be known anyway.

Structure of the Tool Locator

The Tool Locator requires a few fixed RAM locations and no fixed ROM locations. All functions are accessed through the Tool Locator via their tool set number and function number. The Tool Locator uses the tool set number to find an entry in the tool pointer table (TPT). This table contains pointers to function pointer tables (FPTs). Each tool set has an FPT that contains pointers to the individual routines in the tool. The Tool Locator uses the function number to find the address of the routine being called.

Each tool in ROM has an FPT in ROM. In ROM, there is also a TPT that points to all the FPTs in ROM. One fixed RAM location is used to point to this TPT in ROM. This location is initialized at power up and warm boot by the firmware. In this way, the address of the TPT in ROM does not ever have to be fixed.

The TPT has the form shown in Table A-1.

Table A-1
Structure of a TPT (tool pointer table)

Item	Length	Description
Count	4 bytes	Number of tool sets plus 1
Pointer to TS 1 FPT	4 bytes	Pointer to FPT for tool set number 1
Pointer to TS 2 FPT	4 bytes	Pointer to FPT for tool set number 2
...	...	...
Pointer to TS *n* FPT	4 bytes	Pointer to FPT for tool set number *n*

The FPT has the form shown in Table A-2.

Table A-2
Structure of an FPT (function pointer table)

Item	Length	Description
Count	4 bytes	Number of routines plus 1
Address of F1 –1	4 bytes	Pointer to BootInit routine minus 1
Address of F2 –1	4 bytes	Pointer to StartUp routine minus 1
Address of F3 –1	4 bytes	Pointer to ShutDown routine minus 1
Address of F4 –1	4 bytes	Pointer to Version routine minus 1
Address of F5 –1	4 bytes	Pointer to Reset routine minus 1
Address of F6 –1	4 bytes	Pointer to Status routine minus 1
Address of F7 –1	4 bytes	Pointer to reserved routine minus 1
Address of F8 –1	4 bytes	Pointer to reserved routine minus 1
Address of F9 –1	4 bytes	Pointer to first nonrequired routine minus 1
...	...	...
Address of F*n* - 1	4 bytes	Pointer to last nonrequired routine minus 1

Tool set numbers and function numbers

Each system tool is assigned a permanent tool number. Assignment starts at one and continues with each successive integer.

Each function within a tool set is assigned a permanent function number. For the functions within each tool, assignment starts at 1 and continues with each successive integer. Thus, each function has a unique, permanent identifier of the form (*tsNum,funcNum*). Both *tsNum* and *funcNum* are 8-bit numbers. The tool set numbers assigned to the Apple tools are shown in Table 24-2 in Chapter 24, "Tool Locator."

For each tool set, certain standard routines must be present. Each tool set must have a boot initialization routine that is executed at boot time either by the ROM startup code or when the tool set is installed in the system. In addition, each tool set has an application StartUp routine, an application ShutDown routine to allow an application to turn each tool on and off, a Version routine that returns information about the version of the tool, a Reset routine to be called when the system is reset, and a Status routine to indicate whether the tool set is active.

All tools must return version information in the form of a word. The high byte of the word indicates the major release number (starting with 1). The low byte of the word indicates the minor release number (starting with 0). The most significant bit of the word indicates whether the code is an official release or a prototype (no distinction is made between alpha, beta, or other prototype releases).

The standard routines are summarized in Table A-3.

Table A-3
Standard tool set routine numbers

FuncNum	Description
1	Boot initialization function for each tool set
2	Application startup function for each tool set
3	Application shutdown function for each tool set
4	Version information
5	Reset
6	Status
7	Reserved for future use
8	Reserved for future use

Obtaining memory

Tool sets are to obtain any memory they need dynamically (using as little fixed memory as possible) through the Memory Manager. To do that, a tool set needs some way to find the location of its data structures. The Tool Locator maintains a table of work area pointers for the individual tools. The work area pointer table (WAPT) is a table of pointers to the work areas of individual tools.

In the WAPT, each tool will have an entry for its own use. Entries are assigned by tool set number (tool 04 has entry 04 and so on). A pointer to the WAPT is kept at a fixed memory location in RAM so that space for the table can be allocated dynamically.

The Tool Locator system permanently reserves some space in bank $E1 for the purposes shown in Table A-4.

Table A-4
Tool Locator permanent RAM space

Address	Length	Description
$E103C0	4 bytes	Pointer to the active TPT. The pointer is to the ROM-based TPT if there are no RAM-based tool sets and no RAM-based ROM patches. Otherwise, it will point to a RAM-based TPT.
$E103C4	4 bytes	Pointer to the active user's TPT. This pointer is 0 initially, indicating that no user tools are present.
$E103C8	4 bytes	Pointer to the WAPT. The WAPT parallels the TPT. Each WAPT entry is a pointer to a work area assigned to the corresponding tool set. At startup time, each WAPT entry is set to 0, indicating no assigned work area.
$E103CC	4 bytes	Pointer to the user's Work Area Pointer Table (WAPT).
$E10000	16 bytes	Entry points to the dispatcher.

This is the only RAM permanently reserved by the Tool Locator system.

Tool Locator system initialization

Each tool set is initialized before use by application programs. Two types of initialization are needed: boot initialization and application initialization. Boot initialization occurs either at system startup time (boot time) or, for tool sets loaded from disk, when the tool is installed. Regardless of the applications to be executed, the system calls the boot initialization function of every tool set. Thus, each tool set must have a boot initialization routine (*funcNum* = 1), even if it does nothing. This function has no input or output parameters.

Application initialization occurs during application execution. The application calls the application startup function (*funcNum* = 2) of each tool set it will use. The application startup function performs the chores needed to start up the tool set so the application can use it. This function may have inputs and outputs, as defined by the individual tool set.

The application shutdown function (*funcNum* = 3) should be executed as soon as the application no longer needs to use the tool. The shutdown releases the resources used by the tool. As a precaution against applications forgetting to execute the shutdown function, the startup function should either execute the shutdown function itself or do something else to ensure a reasonable startup state.

The provision of two initialization times reflects the needs of currently envisioned tools. On the one hand, for example, the Memory Manager requires boot time initialization because it must operate properly even before any application has been loaded. On the other hand, SANE needs to be initialized only if the system executes some application or desk accessory that uses it. Initializing only the tool sets that will be used saves resources, particularly RAM.

Disk and RAM structure of tool sets

System tool sets are load files kept in the TOOLS subdirectory of the SYSTEM directory. Their file type is $BA; each tool set begins with a function pointer table.

User tool sets may be in any form; it is the responsibility of the application to properly load and install such tool sets.

Installing your tool set

Before you make any calls to a user tool set, you must install it into the system. You do this by calling the Tool Locator routine SetTSPtr. SetTSPtr takes three inputs on the stack as follows:

Stack before SetTSPtr

previous contents	
userOrSystem	**Word**—$0000 = system tool set, $8000 = user tool set
tsNum	**Word**—Tool set number of the tool set
— *fptPtr* —	**Long**—POINTER to function pointer table for tool set
	← **SP**

When SetTSPtr is called, your tool is installed in the system and its boot initialization function call is executed. The following example illustrates installation of a sample user tool:

```
;-------------------------------------------------------------------------------
Install         START

                clc                     ; Switch to full native mode and
                xce                     ; save initial state
                php

                rep #$30                ; 16-bit registers

                PushWord $8000          ; Signal a user tool
                PushWord #$23           ; Put the tool number on the stack
                PushLong #CallTable     ; Point to call table
                _SetTSPtr

                plp                     ; Restore machine state
                xce
                rts
                END
```

```
;------------------------------------------------------------------------------
CallTable      START

               long  (TheEnd-CallTable)/4

               long MyBootInit-1
               long MyStartUp-1
               long MyShutDown-1
               long MyVersion-1
               long MyReset-1
               long NotImp-1
               long NotImp-1
               long NotImp-1

               long FirstFunc-1
               long LastFunc-1

TheEND

               END
;------------------------------------------------------------------------------
MyBootInit     START                   ; Called when installed

               lda #0
               clc
               rtl
               END
;------------------------------------------------------------------------------
MyStartUp      START                   ; User passes me word containing location
                                       ; to use in bank zero
RTL1           equ 1
RTL2           equ RTL1+3
ZPToUse        equ RTL2+3
ToolNum        equ 5                   ; Will be SP+5 just before SetWAP call

               lda ZPToUse,s           ; Get caller's value
               pea $8000               ; Get ready to modify user tool tables
               phx                     ; Tool set number and function number -
                                       ; function number will be erased later
               pea 0                   ; High-order word is zero
               pha                     ; Low-order word is caller's value
               lda ToolNum,s           ; Get tool set number and function number
               and #$00FF              ; Eliminate function number (MSB)
               sta ToolNum,s           ; Parameter now contains only tool set number
               _SetWAP                 ; Set our work area pointer
```

```
; Next, the input parameter must be removed from the stack.
; Do this by sliding the two return addresses up the stack by two bytes,
; such that the most-significant word of the second-level return address
; slides right into the spot previously occupied by the input parameter.
; This isn't very difficult, because the two addresses occupy the same
; number of words.
               lda RTL2+1,s            ; Catch upper two bytes of RTL2
               sta ZPToUse,s           ; Write them over input parameter space
               lda RTL1+2,s            ; Catch MSB of RTL1 and LSB of RTL2
               sta ZPToUse-2
               lda RTL1
               sta RTL1+2
; The bytes at SP+1 and SP+2 (or RTL1 and RTL1+1) are empty and may be pulled
               pla                     ; as a single word!

               lda #0                  ; Report that no error occurred
               clc
               rtl                     ; Done
               END
;-------------------------------------------------------------------------
MyShutDown     START
               cmp #0
               beq Nevermind
               pea $8000               ; Clear out the WAPT entry
               txa                     ; Tool set number and function number were
                                       ; in X on entry
               and #$00FF              ; Eliminate function number (MSB)
               pha                     ; Tell SetWap which entry to clear
               pea 0                   ; and
               pea 0                   ; zero the entry
               _SetWAP

Nevermind      ANOP
               lda #0
               clc
               rtl
               END
;-------------------------------------------------------------------------
MyVersion      START
RTL1           equ 1
RTL2           equ RTL1+3
VerNum         equ RTL2+3

               lda #$90                ; Version 1.0 prototype
               sta VerNum,s
               lda #0
               clc
               rtl
               END
```

```
;-------------------------------------------------------------------------------
MyReset         START

                lda #0
                clc
                rtl
                END
;-------------------------------------------------------------------------------
NotImp          START
                txa                     ; Tool set number and function number were
                                        ; in X on entry
                xba                     ; Tool set number was in LSB, move to MSB
                                        ; for error
                ora #$00FF              ; Easy way to put $FF in LSB as error code
                sec                     ; Raise error flag
                rtl
                END
;-------------------------------------------------------------------------------
FirstFunc       START

                lda #0
                clc
                rtl

                END
;-------------------------------------------------------------------------------
LastFunc        START

                lda #0
                clc
                rtl

                END
;-------------------------------------------------------------------------------
; Notes
;
; The long macro deposits a 4-byte value in memory, low bytes first.
; The PushWord macro pushes a word onto the stack (either from a memory
;               location or with a pea instruction if # is used).
; The PushLong macro pushes a long on the stack (either from memory
;               or with two pea instructions if # is used).
```

Function execution environment

When your function is called, the machine is in full native mode and the following three registers are set with specific information to make the function's job easier:

- A register = low-order word of entry in WAPT for tool
- Y register = high-order word of entry in WAPT for tool
- X register = function number and tool set number

When the function is called, the stack looks like this:

previous contents		
outputSpace		**Word or Long**—Space for output
input1		**Word or Long**—First input
-- *input2* --		**Word or Long**—Second input
inputLast		**Word or Long**—Last input
RTL	*RTL*	**3 bytes**—RTL address
RTL	← **SP**	

Appendix B

Tool Set Error Codes

In Appendix B, the tool set error codes are listed and summarized in error number order. If a tool set error has not occurred, the carry flag (c flag) will not be set and the accumulator will contain $0000.

Table B-1
Tool set error codes

Code	Name	Description
System failure codes		
$0001	pdosUnClmdIntErr	Unclaimed interrupt (ProDOS 16)
$0004	divByZeroErr	Division by 0
$000A	pdosVCBErr	Volume control block unusable (ProDOS 16)
$000B	pdosFCBErr	File control block unusable (ProDOS 16)
$000C	pdosBlk0Err	Block zero allocated illegally (ProDOS 16)
$000D	pdosIntShdwErr	Interrupt with I/O shadowing off (ProDOS 16)
$0015	segLoader1Err	Segment Loader error
$0017	sPackage0Err	Can't load a package
$0018	package1Err	Can't load a package
$0019	package2Err	Can't load a package
$001A	package3Err	Can't load a package
$001B	package4Err	Can't load a package
$001C	package5Err	Can't load a package
$001D	package6Err	Can't load a package
$001E	package7Err	Can't load a package
$0020	package8Err	Can't load a package
$0021	package9Err	Can't load a package

(continued)

Table B-1 (continued)
Tool set error codes

Code	Name	Description
System failure codes		
$0022	`package10Err`	Can't load a package
$0023	`package11Err`	Can't load a package
$0024	`package12Err`	Can't load a package
$0025	`outOfMemErr`	Out of memory
$0026	`segLoader2Err`	Segment Loader error
$0027	`fMapTrshdErr`	File map destroyed
$0028	`stkOvrFlwErr`	Stack overflow
$0030	`psInstDiskErr`	Please insert disk (File Manager alert)
$0032–53		Memory Manager errors
$0100	`stupVolMntErr`	Can't mount system startup volume
Tool Locator codes		
$0001	`toolNotFoundErr`	Specified tool set not found
$0002	`funcNotFoundErr`	Specified routine not found
$0110	`toolVersionErr`	Specified minimum version not found
$0111	`messNotFoundErr`	Specified message not found
Memory Manager codes		
$0201	`memErr`	Unable to allocate block
$0202	`emptyErr`	Illegal operation on an empty handle
$0203	`notEmptyErr`	Empty handle expected for this operation
$0204	`lockErr`	Illegal operation on a locked or immovable block
$0205	`purgeErr`	Attempt to purge an unpurgeable block
$0206	`handleErr`	Invalid handle
$0207	`idErr`	Invalid user ID
$0208	`attrErr`	Illegal operation with specified attributes
Miscellaneous Tool Set codes		
$0301	`badInputErr`	Bad input parameter
$0302	`noDevParamErr`	No device for input parameter
$0303	`taskInstlErr`	Specified task already in Heartbeat queue
$0304	`noSigTaskErr`	No signature detected in task header
$0305	`queueDmgdErr`	Damaged Heartbeat queue detected
$0306	`taskNtFdErr`	Specified task not in queue
$0307	`firmTaskErr`	Unsuccessful firmware task
$0308	`hbQueueBadErr`	Damaged HeartBeat queue detected
$0309	`unCnctdDevErr`	Dispatch attempted to unconnected device
$030B	`idTagNtAvlErr`	No ID tag available

Table B-1 (continued)
Tool set error codes

Code	Name	Description
QuickDraw II codes		
$0401	`alreadyInitialized`	QuickDraw II already initialized
$0402	`cannotReset`	Never used
$0403	`notInitialized`	QuickDraw II not initialized
$0410	`screenReserved`	Screen reserved
$0411	`badRect`	Invalid rectangle specified
$0420	`notEqualChunkiness`	Chunkiness not equal
$0430	`rgnAlreadyOpen`	Region already open
$0431	`rgnNotOpen`	Region not open
$0432	`rgnScanOverflow`	Region scan overflow
$0433	`rgnFull`	Region full
$0440	`polyAlreadyOpen`	Polygon already open
$0441	`polyNotOpen`	Polygon not open
$0442	`polyTooBig`	Polygon too big
$0450	`badTableNum`	Invalid color table number
$0451	`badColorNum`	Invalid color number
$0452	`badScanLine`	Invalid scan line number
$04FF	Not implemented	
Desk Manager codes		
$0510	`daNotFound`	Specified DA not available
$0511	`notSysWindow`	Window pointer is not a pointer to a window owned by an NDA
Event Manager codes		
$0601	`emDupStrtUpErr`	EMStartUp already called
$0602	`emResetErr`	Can't reset Event Manager
$0603	`emNotActErr`	Event Manager not active
$0604	`emBadEvtCodeErr`	Event code is greater than 15
$0605	`emBadBttnNoErr`	Button number specified is not 0 or 1
$0606	`emQSiz2LrgErr`	Size of event queue is greater than 3639
$0607	`emNoMemQueueErr`	Insufficient memory available for queue
$0681	`emBadEvtQErr`	Event queue damaged—fatal system error
$0682	`emBadQHndlErr`	Queue handle damaged—fatal system error
Sound Tool Set codes		
$0810	`noDOCFndErr`	No DOC or RAM found
$0811	`docAddrRngErr`	DOC address range error
$0812	`noSAppInitErr`	No SoundStartUp call made
$0813	`invalGenNumErr`	Invalid generator number
$0814	`synthModeErr`	Synthesizer mode error
$0815	`genBusyErr`	Generator already in use
$0817	`mstrIRQNotAssgnErr`	Master IRQ not assigned
$0818	`sndAlreadyStrtErr`	Sound Tool Set already started
$08FF	`unclaimedSndIntErr`	Unclaimed sound interrupt error (reported through System Failure Manager)

(continued)

Table B-1 (continued)
Tool set error codes

Code	Name	Description
Apple Desktop Bus Tool Set codes		
$0910	`cmndIncomplete`	Command not completed
$0911	`cantSync`	Can't synchronize with system
$0982	`adbBusy`	ADB busy (command pending)
$0983	`devNotAtAddr`	Device not present at address
$0984	`srqListFull`	SRQ list full
Integer Math Tool Set codes		
$0B01	`imBadInptParam`	Bad input parameter
$0B02	`imIllegalChar`	Illegal character in string
$0B03	`imOverflow`	Integer or Longint overflow
$0B04	`imStrOverflow`	String overflow
Text Tool Set codes		
$0C01	`badDevType`	Illegal device type
$0C02	`badDevNum`	Illegal device number
$0C03	`badMode`	Illegal operation
$0C04	`unDefHW`	Undefined hardware error
$0C05	`lostDev`	Lost device: device no longer on-line
$0C06	`lostFile`	File no longer in diskette directory
$0C07	`badTitle`	Illegal filename
$0C08	`noRoom`	Insufficient space on specified diskette
$0C09	`noDevice`	Specified volume not on-line
$0C0A	`noFile`	Specified file not in directory of specified volume
$0C0B	`dupFile`	Duplicate file: attempt to rewrite a file when a file of that name already exists
$0C0C	`notClosed`	Attempt to open file that is already open
$0C0D	`notOpen`	Attempt to access a closed file
$0C0E	`badFormat`	Error in reading real or integer number
$0C0F	`ringBuffOFlo`	Ring buffer overflow: characters arriving faster than the input buffer can accept them
$0C10	`writeProtected`	Specified diskette is write-protected
$0C40	`devErr`	Device error: device failed to complete a read or write correctly
Window Manager codes		
$0E01	`paramLenErr`	First word of parameter list is the wrong size
$0E02	`allocateErr`	Unable to allocate window record
$0E03	`taskMaskErr`	Reserved bits not clear in *wmTaskMask* field of WmTaskRec
Control Manager codes		
$1001	`wmNotStartedUp`	Window Manager not initialized

Table B-1 (continued)
Tool set error codes

Code	Name	Description
Print Manager codes		
$1301	`missingDriver`	Specified driver not in the DRIVERS subdirectory of the SYSTEM subdirectory
$1302	`portNotOn`	Specified port not selected in the control panel
$1303	`noPrintRecord`	No print record specified
$1304	`badLaserPrep`	Version of LaserPrep file in LaserWriter is not compatible with this version of Print Manager
$1305	`badLPFile`	Version of LaserPrep file in DRIVERS subdirectory of SYSTEM subdirectory is not compatible with this version of Print Manager
$1306	`papConnNotOpen`	Connection can't be established with the LaserWriter
$1307	`papReadWriteErr`	Read-write error on the LaserWriter
$1321	`startUpAlreadyMade`	LLDStartUp call already made
$1322	`invalidCtlVal`	Invalid control value specified
LineEdit Tool Set codes		
$1401	`leDupStrtUpErr`	LEStartUp already called
$1402	`leResetError`	Can't reset LineEdit
$1403	`leNotActiveErr`	LineEdit not active
$1404	`leScrapErr`	Desk scrap too big to copy
Dialog Manager codes		
$150A	`badItemType`	Inappropriate item type
$150B	`newItemFailed`	Item creation failed
$150C	`itemNotFound`	No such item
$150D	`notModalDialog`	Frontmost window not a modal dialog window
Scrap Manager codes		
$1610	`badScrapType`	No scrap of this type
Font Manager codes		
$1B01	`fmDupStartUpErr`	FMStartUp call already made
$1B02	`fmResetErr`	Can't reset the Font Manager
$1B03	`fmNotActiveErr`	Font Manager not active
$1B04	`fmFamNotFndErr`	Family not found
$1B05	`fmFontNtFndErr`	Font not found
$1B06	`fmFontMemErr`	Font not in memory
$1B07	`fmSysFontErr`	System font cannot be purgeable
$1B08	`fmBadFamNumErr`	Illegal family number
$1B09	`fmBadSizeErr`	Illegal font size
$1B0A	`fmBadNameErr`	Illegal name length
$1B0B	`fmMenuErr`	FixFontMenu never called
$1B0C	`fmScaleSizeErr`	Scaled size of font exceeds limits

Appendix C

Tool Set Dependencies and Startup Order

In Appendix C, the interdependencies of the tool sets are listed, and the order in which the tool sets must be started up is given.

Tool set dependencies

The Tool Locator (tool set number $01) does not depend on the presence of any of the other tool sets; rather, all of the other tool sets depend on the Tool Locator. Thus, Table C-1 begins with the Memory Manager (tool set number $02) and continues in tool set number order.

❖ *Note:* The dependencies given in Table C-1 differ in some minor respects from those given in the individual tool set chapters. Those in Table C-1 are the most current at the time of publication; they were up to date as of October 1, 1987. For any further updates on the dependencies between the tool sets, check Apple IIGS Technical Note #12.

Table C-1
Tool set dependencies

Tool set number		Tool set name	Minimum version needed
Memory Manager (tool set number $02) depends on			
$01	#01	Tool Locator	1.0
Miscellaneous Tool Set (tool set number $03) depends on			
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0
QuickDraw II (tool set number $04) depends on			
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0
$03	#03	Miscellaneous Tool Set	1.0
Desk Manager (tool set number $05) depends on			
$01	#01	Tool Locator	1.2
$02	#02	Memory Manager	1.2
$03	#03	Miscellaneous Tool Set	1.2
$04	#04	QuickDraw II	1.2
$06	#06	Event Manager	1.0
$0E	#14	Window Manager	1.3
$0F	#15	Menu Manager	1.3
$10	#16	Control Manager	1.3
$14	#20	LineEdit Tool Set	1.0
$15	#21	Dialog Manager	1.0
$16	#22	Scrap Manager	1.0
Event Manager (tool set number $06) depends on			
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0
$03	#03	Miscellaneous Tool Set	1.0
Scheduler (tool set number $07) depends on			
$01	#01	Tool Locator	1.0
$03	#03	Miscellaneous Tool Set	1.0
$02	#02	Memory Manager	1.0
$03	#03	Miscellaneous Tool Set	1.0
Sound Tool Set (tool set number $08) depends on			
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0
$03	#03	Miscellaneous Tool Set	1.0
Apple Desktop Bus Tool Set (tool set number $09) depends on			
$01	#01	Tool Locator	1.0

Table C-1 (continued)
Tool set dependencies

Tool set number		Tool set name	Minimum version needed
SANE Tool Set (tool set number $0A) depends on			
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0
Integer Math Tool Set (tool set number $0B) depends on			
$01	#01	Tool Locator	1.0
Text Tool Set (tool set number $0C) depends on			
$01	#01	Tool Locator	1.0
Window Manager (tool set number $0E) depends on			
$01	#01	Tool Locator	1.2
$02	#02	Memory Manager	1.2
$03	#03	Miscellaneous Tool Set	1.2
$04	#04	QuickDraw II	1.2
$06	#06	Event Manager	1.0
Menu Manager (tool set number $0F) depends on			
$01	#01	Tool Locator	1.2
$02	#02	Memory Manager	1.2
$03	#03	Miscellaneous Tool Set	1.2
$04	#04	QuickDraw II	1.2
$06	#06	Event Manager	1.0
$0E	#14	Window Manager	1.3
$10	#16	Control Manager	1.3
Control Manager (tool set number $10) depends on			
$01	#01	Tool Locator	1.2
$02	#02	Memory Manager	1.2
$03	#03	Miscellaneous Tool Set	1.2
$04	#04	QuickDraw II	1.2
$06	#06	Event Manager	1.0
$0E	#14	Window Manager	1.3
QuickDraw II Auxiliary (tool set number $12) depends on			
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0
$03	#03	Miscellaneous Tool Set	1.0
$04	#04	QuickDraw II	1.2

(continued)

Table C-1 (continued)
Tool set dependencies

Tool set number		Tool set name	Minimum version needed
Print Manager (tool set number $13) depends on			
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	2.0
$03	#03	Miscellaneous Tool Set	2.0
$04	#04	QuickDraw II	2.0
$06	#06	Event Manager	1.0
$0E	#14	Window Manager	1.3
$0F	#15	Menu Manager	1.3
$10	#16	Control Manager	1.3
$12	#18	QuickDraw II Auxiliary	1.0
$14	#20	LineEdit Tool Set	1.0
$15	#21	Dialog Manager	1.1
$1B	#27	Font Manager	1.0
$1C	#28	List Manager	1.0
LineEdit Tool Set (tool set number $14) depends on			
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0
$03	#03	Miscellaneous Tool Set	1.0
$04	#04	QuickDraw II	1.1
$06	#06	Event Manager	1.0

❖ *Note:* If you are going to use the LEToScrap or LEFromScrap routines, the Scrap Manager must be loaded and started up. If you are going to use LETextBox2, you must load and start up the Integer Math Tool Set.

Tool set number		Tool set name	Minimum version needed
Dialog Manager (tool set number $15) depends on			
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0
$03	#03	Miscellaneous Tool Set	1.0
$04	#04	QuickDraw II	1.0
$06	#06	Event Manager	1.0
$0E	#14	Window Manager	1.3
$0F	#15	Menu Manager	1.3
$10	#16	Control Manager	1.3
$14	#20	LineEdit Tool Set	1.0
Scrap Manager (tool set number $16) depends on			
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0

Table C-1 (continued)
Tool set dependencies

Tool set number		Tool set name	Minimum version needed
Standard File Tool Set (tool set number $17) depends on			
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0
$03	#03	Miscellaneous Tool Set	1.0
$04	#04	QuickDraw II	1.0
$06	#06	Event Manager	1.0
$0E	#14	Window Manager	1.3
$10	#16	Control Manager	1.3
$0F	#15	Menu Manager	1.3
$14	#20	LineEdit Tool Set	1.0
$15	#21	Dialog Manager	1.1
Font Manager (tool set number $1B) depends on			
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0
$04	#04	QuickDraw II	1.1

❖ *Note:* In addition to these tool sets, the ChooseFont routine requires

$03	#03	Miscellaneous Tool Set	1.2
$0B	#11	Integer Math Tool Set	1.0
$0E	#14	Window Manager	1.3
$10	#16	Control Manager	1.3
$14	#20	LineEdit Tool Set	1.0
$15	#21	Dialog Manager	1.0
$1C	#28	List Manager	1.0

and the FixFontMenu routine requires

$0F	#15	Menu Manager	1.3
$1C	#28	List Manager	1.0

If you are using the shadowed, outlined, or underlined text styles, QuickDraw II Auxiliary (tool set number $12) must be loaded and started up.

List Manager (tool set number $1C) depends on			
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	1.0
$03	#03	Miscellaneous Tool Set	1.0
$04	#04	QuickDraw II	1.0
$06	#06	Event Manager	1.0
$0E	#14	Window Manager	1.3
$10	#16	Control Manager	1.3

Tool set startup order

Because each tool set depends on the presence of other tool sets, certain tool sets must be started up in a prescribed order for others to work. This order is shown in Table C-2, with tool sets lower on the list depending on the presence of all the tool sets higher on the list. Thus, all tool sets from the Tool Locator through the Control Manager must be started up before the Menu Manager.

When you shut the tools down before you quit your application, you must shut them down in the reverse order from that in which they were started up; that is, the last one started up must be shut down first, the next-to-last started up shut down next, and so on.

❖ *Note:* The startup order given in Table C-2 differs in some minor respects from that given in Chapter 2. Use the order given in Table C-2. For further updates, refer to Apple IIGS Technical Note #12.

Table C-2
Tool set startup order

Tool set number		Tool set name
$01	#01	Tool Locator
$02	#02	Memory Manager
$03	#03	Miscellaneous Tool Set
$04	#04	QuickDraw II
$06	#06	Event Manager
$0E	#14	Window Manager
$10	#16	Control Manager
$0F	#15	Menu Manager
$14	#20	LineEdit Tool Set
$15	#21	Dialog Manager
$17	#23	Standard File Operations Tool Set
$16	#22	Scrap Manager
$05	#05	Desk Manager
$1C	#28	List Manager
$1B	#27	Font Manager
$13	#19	Print Manager

Note: If you are using QuickDraw II Auxiliary, it must be started up after QuickDraw II.

You may assume that tool sets other than those listed in Table C-2 do not need to be started up in any particular order; that is, they may be started up or shut down at any time.

Appendix D

List of Routines by Tool Set Number and Routine Number

This appendix lists the tool set routines in tool set number and routine number order. The last two digits of the heaxadecimal number comprise the tool set number; the first two digits comprise the routine number. Thus, the number $0410 means the routine numbered $04 in the tool set numbered $10.

The list provided in this appendix can be useful, for example, when you are using a debugger and obtain the hexadecimal number, but not the name, of the routine. You can find the name by using this list and then looking up the name in the appropriate chapter.

Table D-1
Routines by tool set/routine number

Number	Routine
Tool Locator (Chapter 24)	
$0101	TLBootInit
$0201	TLStartUp
$0301	TLShutDown
$0401	TLVersion
$0501	TLReset
$0601	TLStatus
$0901	GetTSPtr
$0A01	SetTSPtr
$0B01	GetFuncPtr
$0C01	GetWAP
$0D01	SetWAP
$0E01	LoadTools
$0F01	LoadOneTool
$1001	UnloadOneTool
$1101	TLMountVolume
$1201	TLTextMountVolume
$1301	SaveTextState
$1401	RestoreTextState
$1501	MessageCenter
Memory Manager (Chapter 12)	
$0102	MMBootInit
$0202	MMStartUp
$0302	MMShutDown
$0402	MMVersion
$0502	MMReset
$0602	MMStatus
$0902	NewHandle
$0A02	ReallocHandle
$0B02	RestoreHandle
$1002	DisposeHandle
$1102	DisposeAll
$1202	PurgeHandle
$1302	PurgeAll
$1802	GetHandleSize
$1902	SetHandleSize
$1A02	FindHandle
$1B02	FreeMem
$1C02	MaxBlock
$1D02	TotalMem
$1E02	CheckHandle
$1F02	CompactMem
$2002	HLock
$2102	HLockAll
$2202	HUnlock
$2302	HUnlockAll
$2402	SetPurge
$2502	SetPurgeAll
$2802	PtrToHand
$2902	HandToPtr
$2A02	HandToHand
$2B02	BlockMove
Miscellaneous Tool Set (Chapter 14)	
$0103	MTBootInit
$0203	MTStartUp
$0303	MTShutDown
$0403	MTVersion
$0503	MTReset
$0603	MTStatus
$0903	WriteBRam
$0A03	ReadBRam
$0B03	WriteBParam
$0C03	ReadBParam
$0D03	ReadTimeHex
$0E03	WriteTimeHex
$0F03	ReadAsciiTime
$1003	SetVector
$1103	GetVector
$1203	SetHeartBeat
$1303	DelHeartBeat
$1403	ClrHeartBeat
$1503	SysFailMgr
$1603	GetAddr
$1703	ReadMouse
$1803	InitMouse
$1903	SetMouse
$1A03	HomeMouse
$1B03	ClearMouse
$1C03	ClampMouse
$1D03	GetMouseClamp
$1E03	PosMouse
$1F03	ServeMouse
$2003	GetNewID
$2103	DeleteID
$2203	StatusID
$2303	IntSource
$2403	FWEntry
$2503	GetTick
$2603	PackBytes
$2703	UnPackBytes
$2803	Munger
$2903	GetIRQEnable
$2A03	SetAbsClamp
$2B03	GetAbsClamp
$2C03	SysBeep
QuickDraw II (Chapter 16)	
$0104	QDBootInit
$0204	QDStartUp
$0304	QDShutDown
$0404	QDVersion
$0504	QDReset
$0604	QDStatus
$0904	GetAddress
$0A04	GrafOn
$0B04	GrafOff
$0C04	GetStandardSCB
$0D04	InitColorTable
$0E04	SetColorTable
$0F04	GetColorTable
$1004	SetColorEntry
$1104	GetColorEntry
$1204	SetSCB
$1304	GetSCB
$1404	SetAllSCBs
$1504	ClearScreen
$1604	SetMasterSCB
$1704	GetMasterSCB
$1804	OpenPort
$1904	InitPort

Table D-1 (continued)
Routines by tool set/routine number

Number	Routine
QuickDraw II (Chapter 16)	
$1A04	ClosePort
$1B04	SetPort
$1C04	GetPort
$1D04	SetPortLoc
$1E04	GetPortLoc
$1F04	SetPortRect
$2004	GetPortRect
$2104	SetPortSize
$2204	MovePortTo
$2304	SetOrigin
$2404	SetClip
$2504	GetClip
$2604	ClipRect
$2704	HidePen
$2804	ShowPen
$2904	GetPen
$2A04	SetPenState
$2B04	GetPenState
$2C04	SetPenSize
$2D04	GetPenSize
$2E04	SetPenMode
$2F04	GetPenMode
$3004	SetPenPat
$3104	GetPenPat
$3204	SetPenMask
$3304	GetPenMask
$3404	SetBackPat
$3504	GetBackPat
$3604	PenNormal
$3704	SetSolidPenPat
$3804	SetSolidBackPat
$3904	SolidPattern
$3A04	MoveTo
$3B04	Move
$3C04	LineTo
$3D04	Line
$3E04	SetPicSave
$3F04	GetPicSave
$4004	SetRgnSave
$4104	GetRgnSave
$4204	SetPolySave
$4304	GetPolySave
$4404	SetGrafProcs
$4504	GetGrafProcs
$4604	SetUserField
$4704	GetUserField
$4804	SetSysField
$4904	GetSysField
$4A04	SetRect
$4B04	OffsetRect
$4C04	InsetRect
$4D04	SectRect
$4E04	UnionRect
$4F04	PtInRect
$5004	Pt2Rect
$5104	EqualRect
$5204	EmptyRect
$5204	NotEmptyRect
$5304	FrameRect
$5404	PaintRect
$5504	EraseRect
$5604	InvertRect
$5704	FillRect
$5804	FrameOval
$5904	PaintOval
$5A04	EraseOval
$5B04	InvertOval
$5C04	FillOval
$5D04	FrameRRect
$5E04	PaintRRect
$5F04	EraseRRect
$6004	InvertRRect
$6104	FillRRect
$6204	FrameArc
$6304	PaintArc
$6404	EraseArc
$6504	InvertArc
$6604	FillArc
$6704	NewRgn
$6804	DisposeRgn
$6904	CopyRgn
$6A04	SetEmptyRgn
$6B04	SetRectRgn
$6C04	RectRgn
$6D04	OpenRgn
$6E04	CloseRgn
$6F04	OffsetRgn
$7004	InsetRgn
$7104	SectRgn
$7204	UnionRgn
$7304	DiffRgn
$7404	XorRgn
$7504	PtInRgn
$7604	RectInRgn
$7704	EqualRgn
$7804	EmptyRgn
$7904	FrameRgn
$7A04	PaintRgn
$7B04	EraseRgn
$7C04	InvertRgn
$7D04	FillRgn
$7E04	ScrollRect
$7F04	PaintPixels
$8004	AddPt
$8104	SubPt
$8204	SetPt
$8304	EqualPt
$8404	LocalToGlobal
$8504	GlobalToLocal
$8604	Random
$8704	SetRandSeed
$8804	GetPixel
$8904	ScalePt
$8A04	MapPt
$8B04	MapRect
$8C04	MapRgn
$8D04	SetStdProcs
$8E04	SetCursor
$8F04	GetCursorAdr

(continued)

Table D-1 (continued)
Routines by tool set/routine number

Number	Routine
QuickDraw II (Chapter 16)	
$9004	HideCursor
$9104	ShowCursor
$9204	ObscureCursor
$9404	SetFont
$9504	GetFont
$9604	GetFontInfo
$9704	GetFontGlobals
$9804	SetFontFlags
$9904	GetFontFlags
$9A04	SetTextFace
$9B04	GetTextFace
$9C04	SetTextMode
$9D04	GetTextMode
$9E04	SetSpaceExtra
$9F04	GetSpaceExtra
$A004	SetForeColor
$A104	GetForeColor
$A204	SetBackColor
$A304	GetBackColor
$A404	DrawChar
$A504	DrawString
$A604	DrawCString
$A704	DrawText
$A804	CharWidth
$A904	StringWidth
$AA04	CStringWidth
$AB04	TextWidth
$AC04	CharBounds
$AD04	StringBounds
$AE04	CStringBounds
$AF04	TextBounds
$B004	SetArcRot
$B104	GetArcRot
$B204	SetSysFont
$B304	GetSysFont
$B404	SetVisRgn
$B504	GetVisRgn
$B604	SetIntUse
$B704	OpenPicture (Ch. 17)
$B804	PicComment (Ch. 17)
$B904	ClosePicture (Ch. 17)
$BA04	DrawPicture (Ch. 17)
$BB04	KillPicture (Ch. 17)
$BC04	FramePoly
$BD04	PaintPoly
$BE04	ErasePoly
$BF04	InvertPoly
$C004	FillPoly
$C104	OpenPoly
$C204	ClosePoly
$C304	KillPoly
$C404	OffsetPoly
$C504	MapPoly
$C604	SetClipHandle
$C704	GetClipHandle
$C804	SetVisHandle
$C904	GetVisHandle
$CA04	InitCursor
$CB04	SetBufDims
$CC04	ForceBufDims
$CD04	SaveBufDims
$CE04	RestoreBufDims
$CF04	GetFGSize
$D004	SetFontID
$D104	GetFontID
$D204	SetTextSize
$D304	GetTextSize
$D404	SetCharExtra
$D504	GetCharExtra
$D604	PPToPort
$D704	InflateTextBuffer
$D804	GetROMFont
$D904	GetFontLore
Desk Manager (Chapter 5)	
$0105	DeskBootInit
$0205	DeskStartUp
$0305	DeskShutDown
$0405	DeskVersion
$0505	DeskReset
$0605	DeskStatus
$0905	SaveScrn
$0A05	RestScrn
$0B05	SaveAll
$0C05	RestAll
$0E05	InstallNDA
$0F05	InstallCDA
$1105	ChooseCDA
$1305	SetDAStrPtr
$1405	GetDAStrPtr
$1505	OpenNDA
$1605	CloseNDA
$1705	SystemClick
$1805	SystemEdit
$1905	SystemTask
$1A05	SystemEvent
$1B05	GetNumNDAs
$1C05	CloseNDAByWinPtr
$1D05	CloseAllNDAs
$1E05	FixAppleMenu

Table D-1 (continued)
Routines by tool set/routine number

Event Manager (Chapter 7)

Number	Routine
$0106	EMBootInit
$0206	EMStartUp
$0306	EMShutDown
$0406	EMVersion
$0506	EMReset
$0606	EMStatus
$0906	DoWindows
$0A06	GetNextEvent
$0B06	EventAvail
$0C06	GetMouse
$0D06	Button
$0E06	StillDown
$0F06	WaitMouseUp
$1006	TickCount
$1106	GetDblTime
$1206	GetCaretTime
$1306	SetSwitch
$1406	PostEvent
$1506	FlushEvents
$1606	GetOSEvent
$1706	OSEventAvail
$1806	SetEventMask
$1906	FakeMouse

Scheduler (Chapter 19)

Number	Routine
$0107	SchBootInit
$0207	SchStartUp
$0307	SchShutDown
$0407	SchVersion
$0507	SchReset
$0607	SchStatus
$0907	SchAddTask
$0A07	SchFlush

Sound Tool Set (Chapter 21)

Number	Routine
$0108	SoundBootInit
$0208	SoundStartUp
$0308	SoundShutDown
$0408	SoundVersion
$0508	SoundReset
$0608	SoundToolStatus
$0908	WriteRamBlock
$0A08	ReadRamBlock
$0B08	GetTableAddress
$0C08	GetSoundVolume
$0D08	SetSoundVolume
$0E08	FFStartSound
$0F08	FFStopSound
$1008	FFSoundStatus
$1108	FFGeneratorStatus
$1208	SetSoundMIRQV
$1308	SetUserSoundIRQV
$1408	FFSoundDoneStatus

Apple Desktop Bus Tool Set (Chapter 3)

Number	Routine
$0109	ADBBootInit
$0209	ADBStartUp
$0309	ADBShutDown
$0409	ADBVersion
$0509	ADBReset
$0609	ADBStatus
$0909	SendInfo
$0A09	ReadKeyMicroData
$0B09	ReadKeyMicroMem
$0D09	AsyncADBReceive
$0E09	SyncADBReceive
$0F09	AbsOn
$1009	AbsOff
$1109	ReadAbs
$1209	GetAbsScale
$1309	SetAbsScale
$1409	SRQPoll
$1509	SRQRemove
$1609	ClearSRQTable

SANE Tool Set (Chapter 18)

Number	Routine
$010A	SANEBootInit
$020A	SANEStartUp
$030A	SANEShutDown
$040A	SANEVersion
$050A	SANEReset
$060A	SANEStatus
$090A	SANEFP816
$0A0A	SANEDecStr816
$0B0A	SANEElems816

Integer Math Tool Set (Chapter 9)

Number	Routine
$010B	IMBootInit
$020B	IMStartUp
$030B	IMShutDown
$040B	IMVersion
$050B	IMReset
$060B	IMStatus
$090B	Multiply
$0A0B	SDivide
$0B0B	UDivide
$0C0B	LongMul
$0D0B	LongDivide
$0E0B	FixRatio
$0F0B	FixMul
$100B	FracMul
$110B	FixDiv
$120B	FracDiv
$130B	FixRound
$140B	FracSqrt
$150B	FracCos
$160B	FracSin
$170B	FixATan2
$180B	HiWord
$190B	LoWord
$1A0B	Long2Fix
$1B0B	Fix2Long
$1C0B	Fix2Frac

(continued)

Table D-1 (continued)
Routines by tool set/routine number

Integer Math Tool Set (Chapter 9)

Number	Routine
$1D0B	Frac2Fix
$1E0B	Fix2X
$1F0B	Frac2X
$200B	X2Fix
$210B	X2Frac
$220B	Int2Hex
$230B	Long2Hex
$240B	Hex2Int
$250B	Hex2Long
$260B	Int2Dec
$270B	Long2Dec
$280B	Dec2Int
$290B	Dec2Long
$2A0B	HexIt

Text Tool Set (Chapter 23)

Number	Routine
$010C	TextBootInit
$020C	TextStartUp
$030C	TextShutDown
$040C	TextVersion
$050C	TextReset
$060C	TextStatus
$090C	SetInGlobals
$0A0C	SetOutGlobals
$0B0C	SetErrGlobals
$0C0C	GetInGlobals
$0D0C	GetOutGlobals
$0E0C	GetErrGlobals
$0F0C	SetInputDevice
$100C	SetOutputDevice
$110C	SetErrorDevice
$120C	GetInputDevice
$130C	GetOutputDevice
$140C	GetErrorDevice
$150C	InitTextDev
$160C	CtlTextDev
$170C	StatusTextDev
$180C	WriteChar
$190C	ErrWriteChar
$1A0C	WriteLine
$1B0C	ErrWriteLine
$1C0C	WriteString
$1D0C	ErrWriteString
$1E0C	TextWriteBlock
$1F0C	ErrWriteBlock
$200C	WriteCString
$210C	ErrWriteCString
$220C	ReadChar
$230C	TextReadBlock
$240C	ReadLine

Window Manager (Chapter 25)

Number	Routine
$010E	WindBootInit
$020E	WindStartUp
$030E	WindShutDown
$040E	WindVersion
$050E	WindReset
$060E	WindStatus
$090E	NewWindow
$0A0E	CheckUpdate
$0B0E	CloseWindow
$0C0E	Desktop
$0D0E	SetWTitle
$0E0E	GetWTitle
$0F0E	SetFrameColor
$100E	GetFrameColor
$110E	SelectWindow
$120E	HideWindow
$130E	ShowWindow
$140E	SendBehind
$150E	FrontWindow
$160E	SetInfoDraw
$170E	FindWindow
$180E	TrackGoAway
$190E	MoveWindow
$1A0E	DragWindow
$1B0E	GrowWindow
$1C0E	SizeWindow
$1D0E	TaskMaster
$1E0E	BeginUpdate
$1F0E	EndUpdate
$200E	GetWMgrPort
$210E	PinRect
$220E	HiliteWindow
$230E	ShowHide
$240E	BringToFront
$250E	WindNewRes
$260E	TrackZoom
$270E	ZoomWindow
$280E	SetWRefCon
$290E	GetWRefCon
$2A0E	GetNextWindow
$2B0E	GetWKind
$2C0E	GetWFrame
$2D0E	SetWFrame
$2E0E	GetStructRgn
$2F0E	GetContentRgn
$300E	GetUpdateRgn
$310E	GetDefProc
$320E	SetDefProc
$330E	GetWControls
$340E	SetOriginMask
$350E	GetInfoRefCon
$360E	SetInfoRefCon
$370E	GetZoomRect
$380E	SetZoomRect
$390E	RefreshDesktop
$3A0E	InvalRect
$3B0E	InvalRgn
$3C0E	ValidRect
$3D0E	ValidRgn
$3E0E	GetContentOrigin
$3F0E	SetContentOrigin
$400E	GetDataSize
$410E	SetDataSize
$420E	GetMaxGrow
$430E	SetMaxGrow
$440E	GetScroll
$450E	SetScroll
$460E	GetPage
$470E	SetPage

Table D-1 (continued)
Routines by tool set/routine number

Window Manager (Chapter 25)

Number	Routine
$480E	GetContentDraw
$490E	SetContentDraw
$4A0E	GetInfoDraw
$4B0E	SetSysWindow
$4C0E	GetSysWFlag
$4D0E	StartDrawing
$4E0E	SetWindowIcons
$4F0E	GetRectInfo
$500E	StartInfoDrawing
$510E	EndInfoDrawing
$520E	GetFirstWindow
$530E	WindDragRect
$560E	WindowGlobal
$570E	SetContentOrigin2

Menu Manager (Chapter 13)

Number	Routine
$010F	MenuBootInit
$020F	MenuStartUp
$030F	MenuShutDown
$040F	MenuVersion
$050F	MenuReset
$060F	MenuStatus
$090F	MenuKey
$0A0F	GetMenuBar
$0B0F	MenuRefresh
$0C0F	FlashMenuBar
$0D0F	InsertMenu
$0E0F	DeleteMenu
$0F0F	InsertMItem
$100F	DeleteMItem
$110F	GetSysBar
$120F	SetSysBar
$130F	FixMenuBar
$140F	CountMItems
$150F	NewMenuBar
$160F	GetMHandle
$170F	SetBarColors
$180F	GetBarColors
$190F	SetMTitleStart
$1A0F	GetMTitleStart
$1B0F	GetMenuMgrPort
$1C0F	CalcMenuSize
$1D0F	SetMTitleWidth
$1E0F	GetMTitleWidth
$1F0F	SetMenuFlag
$200F	GetMenuFlag
$210F	SetMenuTitle
$220F	GetMenuTitle
$230F	MenuGlobal
$240F	SetMItem
$250F	GetMItem
$260F	SetMItemFlag
$270F	GetMItemFlag
$280F	SetMItemBlink
$290F	MenuNewRes
$2A0F	DrawMenuBar
$2B0F	MenuSelect
$2C0F	HiliteMenu
$2D0F	NewMenu
$2E0F	DisposeMenu
$2F0F	InitPalette
$300F	EnableMItem
$310F	DisableMItem
$320F	CheckMItem
$330F	SetMItemMark
$340F	GetMItemMark
$350F	SetMItemStyle
$360F	GetMItemStyle
$370F	SetMenuID
$380F	SetMItemID
$390F	SetMenuBar
$3A0F	SetMItemName

Control Manager (Chapter 4)

Number	Routine
$0110	CtlBootInit
$0210	CtlStartUp
$0310	CtlShutDown
$0410	CtlVersion
$0510	CtlReset
$0610	CtlStatus
$0910	NewControl
$0A10	DisposeControl
$0B10	KillControls
$0C10	SetCtlTitle
$0D10	GetCtlTitle
$0E10	HideControl
$0F10	ShowControl
$1010	DrawControls
$1110	HiliteControl
$1210	CtlNewRes
$1310	FindControl
$1410	TestControl
$1510	TrackControl
$1610	MoveControl
$1710	DragControl
$1810	SetCtlIcons
$1910	SetCtlValue
$1A10	GetCtlValue
$1B10	SetCtlParams
$1C10	GetCtlParams
$1D10	DragRect
$1E10	GrowSize
$1F10	GetCtlDPage
$2010	SetCtlAction
$2110	GetCtlAction
$2210	SetCtlRefCon
$2310	GetCtlRefCon
$2410	EraseControl
$2510	DrawOneCtl

(continued)

Table D-1 (continued)
Routines by tool set/routine number

Loader (not documented in this manual; see *Apple IIGS ProDOS 16 Reference*)

Number	Routine
$0111	LoaderInitialization
$0211	LoaderStartUp
$0311	LoaderShutDown
$0411	LoaderVersion
$0511	LoaderReset
$0611	LoaderStatus
$0911	InitialLoad
$0A11	Restart
$0B11	LoadSegNum
$0C11	UnloadSegNum
$0D11	LoadSegName
$0E11	UnloadSeg
$0F11	GetLoadSegInfo
$1011	GetUserID
$1111	LGetPathname
$1211	UserShutDown

QuickDraw Auxiliary (Chapter 17)

Number	Routine
$0112	QDAuxBootInit
$0212	QDAuxStartUp
$0312	QDAuxShutDown
$0412	QDAuxVersion
$0512	QDAuxReset
$0612	QDAuxStatus
$0912	CopyPixels
$0A12	WaitCursor
$0B12	DrawIcon

Print Manager (Chapter 15)

Number	Routine
$0113	PMBootInit
$0213	PMStartUp
$0313	PMShutDown
$0413	PMVersion
$0513	PMReset
$0613	PMStatus
$0913	PrDefault
$0A13	PrValidate
$0B13	PrStlDialog
$0C13	PrJobDialog
$0D13	PrPixelMap
$0E13	PrOpenDoc
$0F13	PrCloseDoc
$1013	PrOpenPage
$1113	PrClosePage
$1213	PrPicFile
$1413	PrError
$1513	PrSetError
$1613	PrChoosePrinter
$2313	PrDriverVer
$2413	PrPortVer

LineEdit Tool Set (Chapter 10)

Number	Routine
$0114	LEBootInit
$0214	LEStartUp
$0314	LEShutDown
$0414	LEVersion
$0514	LEReset
$0614	LEStatus
$0914	LENew
$0A14	LEDispose
$0B14	LESetText
$0C14	LEIdle
$0D14	LEClick
$0E14	LESetSelect
$0F14	LEActivate
$1014	LEDeactivate
$1114	LEKey
$1214	LECut
$1314	LECopy
$1414	LEPaste
$1514	LEDelete
$1614	LEInsert
$1714	LEUpdate
$1814	LETextBox
$1914	LEFromScrap
$1A14	LEToScrap
$1B14	LEScrapHandle
$1C14	LEGetScrapLen
$1D14	LESetScrapLen
$1E14	LESetHilite
$1F14	LESetCaret
$2014	LETextBox2
$2114	LESetJust
$2214	LEGetTextHand
$2314	LEGetTextLen

Table D-1 (continued)
Routines by tool set/routine number

Number	Routine
Dialog Manager (Chapter 6)	
$0115	DialogBootInit
$0215	DialogStartUp
$0315	DialogShutDown
$0415	DialogVersion
$0515	DialogReset
$0615	DialogStatus
$0915	ErrorSound
$0A15	NewModalDialog
$0B15	NewModelessDialog
$0C15	CloseDialog
$0D15	NewDItem
$0E15	RemoveDItem
$0F15	ModalDialog
$1015	IsDialogEvent
$1115	DialogSelect
$1215	DlgCut
$1315	DlgCopy
$1415	DlgPaste
$1515	DlgDelete
$1615	DrawDialog
$1715	Alert
$1815	StopAlert
$1915	NoteAlert
$1A15	CautionAlert
$1B15	ParamText
$1C15	SetDAFont
$1E15	GetControlDItem
$1F15	GetIText
$2015	SetIText
$2115	SelIText
$2115	SelectIText
$2215	HideDItem
$2315	ShowDItem
$2415	FindDItem
$2515	UpdateDialog
$2615	GetDItemType
$2715	SetDItemType
$2815	GetDItemBox
$2915	SetDItemBox
$2A15	GetFirstDItem
$2B15	GetNextDItem
$2C15	ModalDialog2
$2E15	GetDItemValue
$2F15	SetDItemValue
$3215	GetNewModalDialog
$3315	GetNewDItem
$3415	GetAlertStage
$3515	ResetAlertStage
$3615	DefaultFilter
$3715	GetDefButton
$3815	SetDefButton
$3915	DisableDItem
$3A15	EnableDItem
Scrap Manager (Chapter 20)	
$0116	ScrapBootInit
$0216	ScrapStartUp
$0316	ScrapShutDown
$0416	ScrapVersion
$0516	ScrapReset
$0616	ScrapStatus
$0916	UnloadScrap
$0A16	LoadScrap
$0B16	ZeroScrap
$0C16	PutScrap
$0D16	GetScrap
$0E16	GetScrapHandle
$0F16	GetScrapSize
$1016	GetScrapPath
$1116	SetScrapPath
$1216	GetScrapCount
$1316	GetScrapState
Standard File Operations Tool Set (Chapter 22)	
$0117	SFBootInit
$0217	SFStartUp
$0317	SFShutDown
$0417	SFVersion
$0517	SFReset
$0617	SFStatus
$0917	SFGetFile
$0A17	SFPutFile
$0B17	SFPGetFile
$0C17	SFPPutFile
$0D17	SFAllCaps
Note Synthesizer (not documented in this manual)	
$0119	NSBootInit
$0219	NSStartUp
$0319	NSShutDown
$0419	NSVersion
$0519	NSReset
$0619	NSStatus
$0919	AllocGen
$0A19	DeallocGen
$0B19	NoteOn
$0C19	NoteOff
$0D19	AllNotesOff

(continued)

Table D-1 (continued)
Routines by tool set/routine number

Number	Routine	Number	Routine	Number	Routine
Note Sequencer (not documented in this manual)		**Font Manager** (Chapter 8)		**List Manager** (Chapter 11)	
$011A	SeqBootInit	$011B	FMBootInit	$011C	ListBootInit
$021A	SeqStartUp	$021B	FMStartUp	$021C	ListStartup
$031A	SeqShutDown	$031B	FMShutDown	$031C	ListShutDown
$041A	SeqVersion	$041B	FMVersion	$041C	ListVersion
$051A	SeqReset	$051B	FMReset	$051C	ListReset
$061A	SeqStatus	$061B	FMStatus	$061C	ListStatus
$091A	SetIncr	$091B	CountFamilies	$091C	CreateList
$0A1A	SeqBootInit	$0A1B	FindFamily	$0A1C	SortList
$0A1A	ClearIncr	$0B1B	GetFamInfo	$0B1C	NextMember
$0B1A	GetTimer	$0C1B	GetFamNum	$0C1C	DrawMember
$0C1A	GetLoc	$0D1B	AddFamily	$0D1C	SelectMember
$0D1A	AllNotesOff	$0E1B	InstallFont	$0E1C	GetListDefProc
$0E1A	SetTrkInfo	$0F1B	SetPurgeStat	$0F1C	ResetMember
$0F1A	StartSeq	$101B	CountFonts	$101C	NewList
$101A	StepSeq	$111B	FindFontStats		
$111A	StopSeq	$121B	LoadFont		
$121A	SetInstTable	$131B	LoadSysFont		
$131A	StartInts	$141B	AddFontVar		
$141A	StopInts	$151B	FixFontMenu		
		$161B	ChooseFont		
		$171B	ItemID2FamNum		
		$181B	FMSetSysFont		
		$191B	FMGetSysFID		
		$1A1B	FMGetCurFID		
		$1B1B	FamNum2ItemID		

Glossary

absolute: Said of a load segment or other program code that must be loaded at a specific address in memory and never moved. Compare **relocatable.**

absolute addressing: An addressing mode in which instruction operands are interpreted as literal addresses.

absolute clamps: Values that establish the minimum and maximum X and Y coordinates for alternative pointing devices.

access, access byte: An attribute of a ProDOS file that controls whether the file may be read from, written to, renamed, or backed up.

accumulator: The register in a computer's central processor or microprocessor where most computations are performed. Also called *A register*.

activate: To make active. A control or window may be activated. Compare **enable.**

activate event: A window event that occurs when a window is made either active or inactive.

active: Able to respond to the user's mouse or keyboard actions. Controls and windows that are active are displayed differently than inactive items.

ADB: See **Apple Desktop Bus.**

ADB commands: Commands that are issued to the Apple Desktop Bus. These are not the same as Apple Desktop Bus Tool Set routines; rather, the tool set routines often include an ADB command as a parameter. The Apple Desktop Bus Tool Set then interprets and issues the ADB command.

alert: A warning or report of an error in the form of an alert box, a sound from the computer's speaker, or both.

alert box: A special type of dialog box that appears on the screen to give a warning or to report an error message during use of an application.

alert sound: A sound generated by a sound procedure that emits a tone or sequence of tones when the user is to be alerted of a condition.

alert stage: One of four stages that correspond to consecutive occurrences of an alert.

alert template: A data structure that conatains an alert ID, a RECT determining the alert window's size and location, information about what should happen at each stage of the alert, and a list of pointers to the item templates.

alert window: The window in which an alert box appears. One of the two predefined window formats. Compare **document window.**

alternative pointing devices: A device, such as a graphics tablet or trackball, used instead of the mouse.

Apple Desktop Bus (ADB): An input bus, with its own protocol and electrical characteristics, that provides a method of connecting input devices such as keyboards and mice with personal computers.

Apple Desktop Bus Tool Set: The Apple IIGS tool set that facilitates an application's interaction with devices connected to the Apple Desktop Bus.

Apple key: A modifier key on the Apple IIGS keyboard, marked with both an Apple icon and a *spinner;* the icon used on the equivalent key on some Macintosh keyboards. It performs the same functions as the open Apple key on standard Apple II machines.

AppleTalk network: A local area network developed by Apple Computer, Inc.

Apple II: A family of computers, including the original Apple II, the Apple II Plus, the Apple IIe, the Apple IIc, and the Apple IIGS. Compare **standard Apple II.**

Apple II Plus: A personal computer in the Apple II family with expansion slots that allow the user to enhance the computer's capabilities with peripheral cards.

Apple IIc: A transportable personal computer in the Apple II family, with a disk drive, serial ports, and 80-column display capability built in.

Apple IIe: A personal computer in the Apple II family, with seven expansion slots and an auxiliary memory slot that allow the user to enhance the computer's capabilities with peripheral memory and video enhancement cards.

Apple IIGS: The most advanced computer in the Apple II family. It features expanded memory, advanced sound and graphics, and the Apple IIGS Toolbox of programming routines.

Apple IIGS Programmer's Workshop: See **APW.**

Apple IIGS Toolbox: An extensive set of routines that facilitates writing desktop applications and provides easy program access to many Apple IIGS hardware and firmware features. Functions within the toolbox are grouped into tool sets.

application: A stand-alone program that performs a specific function, such as word processing, drawing, or telecommunications. Compare, for example, **desk accessory, device driver.**

application event: Any of four types of events available for applications to define and respond to as desired.

application prefix: The ProDOS 16 prefix number `1/`. It specifies the directory of the currently running application.

application window: A window in which an application's document appears.

APW (Apple IIGS Programmer's Workshop): A multilanguage development environment for writing Apple IIGS desktop applications.

APW Assembler: The 65816 assembly-language assembler provided with the Apple IIGS Programmer's Workshop.

APW C Compiler: The C-language compiler provided with the Apple IIGS Programmer's Workshop.

APW Shell: The programming environment of the Apple IIGS Programmer's Workshop. It provides facilities for file manipulation and program execution, and supports shell applications.

arbitrary mode: In the List Manager, a selection mode that allows the user to select members in a list without deselecting already-selected members.

arc: A portion of an oval; one of the fundamental shapes drawn by QuickDraw II.

A register: See **accumulator.**

ascent: In a font, the distance between the base line and the ascent line.

ascent line: A horizontal line that coincides with the tops of the tallest characters in a font. See also **base line, descent line.**

ASCII: Acronym for *American Standard Code for Information Interchange,* pronounced "ASK-ee." A code in which the numbers from 0 to 127 stand for text characters. ASCII code is used to represent text inside a computer and to transmit text between computers or between a computer and a peripheral device.

assembler: A language translator that converts a program written in assembly language into an equivalent program in machine language.

AsyncADBReceive completion routine: Used in conjunction with the ADB Tool Set routine AsyncADBReceive, the completion routine obtains ADB data from a buffer. Compare **SRQ list completion routine.**

attributes word: Determines how memory blocks are allocated and maintained. Most of the attributes are defined at allocation time and can't be changed after that; other attributes can be modified after allocation.

auto-key: A keyboard feature and an event type, in which a key being held down continuously is interpreted as a rapid series of identical keystrokes.

auxID: A subfield of the user ID. An application may place any value it wishes into the *auxID* field.

auxiliary type: A secondary classification of ProDOS files. A file's auxiliary-type field may contain information of use to the applications that read it. Compare **file type.**

available font: A font that the Font Manager can use because the font is the ROM font, or a font in the FONTS subdirectory, or a font that the application has added with the Font Manager routine AddFontVar.

background: The pixels within a character or other screen object that are not part of the object itself.

background color: The color of background pixels in text; by default it is black.

background pattern: The pattern QuickDraw II uses to erase objects on the screen.

background pixels: In a character image, the pixels that are not part of the character itself; that is, all pixels in the character bounds rectangle that are not foreground pixels.

background procedure: A procedure run by the Print Manager whenever the Print Manager has directed output to the printer and is waiting for the printer to finish.

bank: A 64K (65,536-byte) portion of the Apple IIGS internal memory. An individual bank is specified by the value of the 65C816 microprocessor's bank register.

bank $00: The first bank of memory in the Apple IIGS. In emulation mode, it is equivalent to *main memory* in an Apple IIe or Apple IIc computer.

base family: A font family is a base family if it is the ROM font or if a plain-styled example of the family can be found among the fonts in the FONTS subdirectory.

base height: In the LineEdit Tool Set, the distance between the top of the destination rectangle and the base line. This controls where the text is drawn.

base line: A horizontal line that coincides with the bottom of the main body of each character in a font. Character descenders extend below the base line.

BASIC: Acronym for *Beginners All-purpose Symbolic Instruction Code*. BASIC is a high-level programming language designed to be easy to learn.

best-fit font algorithm: The algorithm that the Font Manager routine InstallFont uses to look for a font that matches a given set of specifications.

bit: A contraction of *binary digit,* the smallest representation of data in a digital computer.

bit plane: A method of representing images in computer memory. In a bit plane, consecutive bits in memory specify adjacent pixels in the image; if more than one bit is required to completely specify the state of a pixel, more than one bit plane is used for the image. Compare **chunky pixels.**

block: (1) A unit of data storage or transfer, typically 512 bytes. (2) A contiguous region of computer memory of arbitrary size, allocated by the Memory Manager. Also called *memory block.*

block device: A device that transfers data to or from a computer in multiples of 1 block (512 bytes) of characters at a time. Disk drives are block devices. Also called *block I/O device.*

block I/O device: See **block device.**

Boolean logic: A mathematical system in which every expression evaluates to one of two values, usually referred to as TRUE or FALSE.

Boolean variable: A variable that can have one of two values, usually referred to as TRUE or FALSE.

bottom scroll bar: The scroll bar the user selects to scroll horizontally through the data in a window.

boundary rectangle: A rectangle, defined as part of a QuickDraw II *locInfo* record, that encloses the active area of the pixel image and imposes a coordinate system on it. Its top left corner is always aligned on the first pixel in the pixel map.

boundsRect: The GrafPort field that defines the port's boundary rectangle.

buffer: A holding area of the computer's memory where information can be stored by one program or device and then read, perhaps at a different rate, by another; for example, a print buffer.

busy flag: A feature that informs the Scheduler whether a currently needed resource is busy or available.

button: (1) A pushbutton-like image in dialog boxes where the user clicks to designate, confirm, or cancel an action. See also **check box, radio button.** (2) A button on a mouse or other pointing device. See also **mouse button.**

byte: A unit of information consisting of 8 bits. A byte can have any value between 0 and 255, which may represent an instruction, a letter, a number, a punctuation mark, or another character. See also **bit, kilobyte, megabyte.**

C: A high-level programming language. One of the languages available for the Apple IIGS Programmer's Workshop.

cancel: To stop an operation, such as the setting of page-setup values in a dialog box, without saving any results produced up to that point.

Cancel: One of two predefined item ID numbers for dialog box buttons (Cancel = 2). Compare **OK.**

caret: A symbol that indicates where something should or will be inserted in text. On the screen it designates the insertion point, and is usually a vertical bar (|).

carry flag: A status bit in the microprocessor indicating whether an accumulator calculation has resulted in a carry out of the register. Also called *c flag.*

CDA: See **classic desk accessory.**

CDA menu: The menu on which classic desk accessories are listed; the user selects the menu by pressing Control-Apple-Escape. See also **classic desk accessory.**

c flag: See **carry flag.**

character: (1) Any symbol that has a widely understood meaning and thus can convey information. Some characters—such as letters, numbers, and punctuation—can be displayed on the monitor screen and printed on a printer. Most characters are represented in the computer as 1-byte values. (2) In QuickDraw II, a single ASCII character.

character bounds rectangle: The rectangle that determines the extent of the background pixels of a character.

character bounds width: The width of a character's character bounds rectangle.

character device: A device that transfers data to or from a computer as a stream of individual characters. Keyboards and printers are character devices.

character image: The part of a font strike that represents a character in a font.

character image width: The number of columns in a character image.

character origin: The point on the base line used as a reference location for drawing a character.

character position: An index into LineEdit text, with position 0 corresponding to the first character.

character rectangle: A rectangle that encloses a character image. Its width is equal to the image width of the character; its height is equal to the character height.

character width: The number of pixels the pen position is to be advanced after the character is drawn.

check box: A small box associated with an option in a dialog box. When the user clicks the check box, that may change the option or affect related options. See also **radio button.**

Choose Printer dialog box: A Print Manager dialog box that lets the user select a printer or port for printing.

chunkiness: The number of bits required to describe the state of a pixel in a pixel image.

chunky pixels: A method of representing images in computer memory. In chunky pixel organization, a number of consecutive bits in memory combine to specify the state of a single pixel in the image. Consecutive *groups* of bits (the size of the group is equal to the image's chunkiness) define adjacent pixels in the image. Compare **bit plane.**

clamp values: The X- and Y-limits, in terms of pixels, on cursor position controlled by mouse movement.

classic desk accessory (CDA): Desk accessories designed to execute in a nondesktop-, nonevent-based environment. Compare **new desk accessory.**

click: To position the pointer on something, and then to press and quickly release the mouse or alternative pointing device's button.

clip: To restrict drawing to within a particular boundary; any drawing attempted outside that boundary does not occur.

Clipboard: The holding place for the material the user last cut or copied; a buffer area in memory. Information on the Clipboard can be inserted (pasted) into documents. In memory, the contents of the clipboard are called the *desk scrap*.

clipping region: The region to which an application limits drawing in a GrafPort.

clock: (1) The timing circuit that controls execution of a microprocessor. Also called *system clock*. (2) An integrated circuit, often with battery-backup memory, that gives the current date and time.

close box: The small white box on the left side of the title bar of an active window. Clicking it closes the window.

color table: One table of 16 lookup tables in Apple IIGS memory. The table lists the available color values for a scan line.

compaction: The rearrangement of allocated blocks in memory to open up larger contiguous areas of free space.

compiler: A program that produces object files (containing machine-language code) from source files written in a high-level language such as C. Compare **assembler.**

content region: The area in a window in which your application presents information to the user.

control: An object in a window with which the user, using the mouse, can cause instant action with visible results or change settings to modify a future action.

control definition procedure: A procedure used to define the appearance and behavior of a custom control.

Control Manager: The Apple IIGS tool set that manages controls, which are objects on the screen that the user can manipulate with the mouse to cause instant action or change settings.

Control Panel: A desk accessory that lets the user change certain system parameters, such as speaker volume, display colors, and configuration of slots and ports.

control record: A data structure that defines the appearance and behavior of a control.

coordinate plane: A two-dimensional grid defined by QuickDraw II. All drawing commands are located in terms of coordinates on the grid.

coordinates: X and Y locations on the QuickDraw II coordinate plane. Most QuickDraw II routines accept and return coordinates in the order (Y,X).

copy: To duplicate something by selecting it and choosing Copy from the Edit menu. A copy of the selected portion is placed on the Clipboard, without affecting the original selection.

C string: An ASCII character string terminated by a null character (ASCII value = 0). Compare **Pascal string.**

C-type string: Same as *C string*.

current font: The font currently being used by QuickDraw II to draw text.

cursor: A symbol displayed on the screen, marking where the user's next action will take effect or where the next character typed from the keyboard will appear.

cursor record: The data structure that defines the height and width of the cursor, the image of the cursor, the mask controlling the appearance of the cursor, and the hot spot defining where the image of the cursor will be placed by the mouse.

cut: To remove something by selecting it and choosing Cut from the Edit menu. The cut portion is placed on the Clipboard.

data area: The name for a document as viewed in a window. The data area is the entire document, only a portion of which (the visible region) may be seen in the window at any one time.

data bank register: A register in the 65C816 processor that contains the high-order byte of the 24-bit address that references data in memory.

data structure: A specifically formatted item of data or a form into which data may be placed.

DB register: See **data bank register.**

dead character: A character with a character width of 0.

default button: The button in a dialog box whose action will be executed if the user presses the Return key.

default prefix: The pathname prefix attached by ProDOS 16 to a partial pathname when no prefix number is supplied by the application. The default prefix is equivalent to prefix number `0/`.

dereference: To substitute a pointer for a memory handle, or a value for a pointer. When you dereference a memory block's handle, you access the block directly (through its master pointer) rather than indirectly (through its handle).

descender: Any part of a character that lies below the base line.

descent: In a font, the distance between the base line and the descent line.

descent line: A horizontal line that coincides with the bottoms of character descenders (such as the tail on a lowercase "p") that extends farthest below the base line. See also **ascent line, base height, font height.**

desk accessory: A "mini-application" that is available to the user regardless of whether another application is running. The Apple IIGS supports two types of desk accessories: classic desk accessories and new desk accessories.

desk accessory event: An event that occurs whenever the user presses Control-Apple-Escape to invoke a classic desk accessory.

Desk Accessory menu: The menu whose title is a colored apple symbol.

Desk Manager: The Apple IIGS tool set that executes desk accessories and enables applications to support them.

desk scrap: A piece of data, maintained by the Scrap Manager, taken from one application and available for insertion into another.

desktop: The visual interface between the computer and the user—the menu bar and the gray (or solid-colored) area on the screen. In many applications the user can have a number of documents on the desktop at the same time.

desktop user interface: See **desktop.**

destination location: The location (memory buffer or portion of the QuickDraw II coordinate plane) to which data such as text or graphics are copied. See also **destination rectangle.**

destination rectangle: (1) The rectangle (on the QuickDraw II coordinate plane) in which text or graphics are drawn when transferred from somewhere else. Compare **source rectangle.** (2) In LineEdit, the rectangle that determines where the text will be drawn.

device: A piece of hardware used in conjunction with a computer and under the computer's control. Also called *peripheral device* because such equipment is often physically separate from, but attached to, the computer.

device driver: A program that handles the transfer of data to and from a peripheral device, such as a printer or disk drive.

device driver event: An event generated by a device driver.

dial: An indicator on the screen that displays a quantitative setting or value. Usually found in analog form, such as a fuel gauge or a thermometer. A scroll bar is a standard type of dial.

dialog: See **dialog box.**

dialog box: A box on the screen that contains a message requesting more information from the user. See also **alert box.**

Dialog Manager: The Apple IIGS tool set that manipulates dialog boxes and alerts, which appear on the screen when an application needs more information to carry out a command or when the user needs to be notified of an important situation.

dialog pointer: A pointer to a dialog's GrafPort; equivalent to the window pointer for the dialog box.

dialog record: Information describing a dialog window that is maintained by the Dialog Manager.

dialog template: A record that contains information about a dialog to be created.

dialog window: The window in which a dialog box appears.

digital oscillator chip (DOC): An integrated circuit in the Apple IIGS that contains 32 digital oscillators, each of which can generate a sound from stored digital waveform data.

dim: On the Apple IIGS desktop, to display a control or menu item in gray rather than black, to notify the user that the item is inactive.

direct page: A page (256 bytes) of bank $00 of Apple IIGS memory, any part of which can be addressed with a short (1-byte) address because its high-order address byte is always $00 and its middle address byte is the value of the 65C816 direct register. Co-resident programs or routines can have their own direct pages at different locations. The direct page corresponds to the 6502 processor's zero page. The term *direct page* is often used informally to refer to any part of the direct-page/stack space. Compare **zero page.**

direct-page/stack segment: A program segment that is used to initialize the size and contents of an application's stack and direct page.

direct-page/stack space: A single block of memory that contains an application's stack and direct page.

direct register: A hardware register in the 65C816 processor that specifies the start of the direct page.

disable: To make unresponsive to user actions. A dialog box control that is disabled does nothing when selected or manipulated by the user. In appearance, however, it is identical to an enabled control. Compare **inactive.**

disabled menu: A menu that can be pulled down, but in which items are dimmed and not selectable.

display mode: A specification for the way in which a video display functions, including such parameters as text or graphics display, available colors, and number of pixels. The Apple IIGS has two text display modes (40 column and 80 column), two standard Apple II graphics display modes (320 mode and 640 mode), and two new Super Hi-Res graphics display modes.

display rectangle: A rectangle that determines where an item is displayed within a dialog box.

dispose: To permanently deallocate (a memory block). The Memory Manager disposes of a memory block by removing its master pointer. Any handle to that pointer will then be invalid. Compare **purge.**

dithering: A technique for alternating the values of adjacent pixels to create the optical effect of intermediate values. Dithering can give the effect of shades of gray on a black-and-white display, or more colors on a color display.

dividing line: A line that divides groups of items in a menu; such a line uses the space of an entire item and requires an item record. Compare **underline.**

DOC: See **digital oscillator chip.**

document: A file created by an application.

document window: A window that displays a document. One of the two predefined window formats. Compare **alert window.**

double-click: To position the pointer where you want an action to take place, and then press and release the mouse button twice in quick succession without moving the mouse.

draft printing: The print method that the LaserWriter uses. QuickDraw II calls are converted directly into command codes the printer understands, which are then immediately used to drive the printer. Compare **spool printing.**

drag: To position the pointer on something, press and hold the mouse button, move the mouse, and release the mouse button. When you release the mouse button, you either confirm a menu selection or move an object to a new location.

drag region: A region in a window (usually on the title bar) in which the mouse pointer must be placed before the user can drag the window.

draw: In QuickDraw II, to color pixels in a pixel image.

drawing environment: The complete description of how and where drawing may take place. Every open window on the Apple IIGS screen is associated with a GrafPort record, which specifies the window's drawing environment. Same as *graphic port, port.*

drawing mask: An 8-bit by 8-bit pattern that controls which pixels in the QuickDraw II pen will be modified when the pen draws.

drawing mode: One of 16 possible interactions between pixels in QuickDraw II's pen pattern and pixels already on the screen that fall under the pen's path. In `modeCopy` mode, for example, pixels already on the screen are ignored. In `modeXOR` mode, on the other hand, bits in pixels on the screen are XOR'd with bits in pixels in the pen; the resulting pixels are drawn on the screen. See also **pen mode, text mode**.

drawing pen: See **pen.**

driver: See **device driver.**

dynamic segment: A load segment capable of being loaded during program execution. Compare **static segment.**

edit record: A complete text-editing environment in the LineEdit Tool Set, which includes the text to be edited, the GrafPort and rectangle in which to display the text, the arrangement of the text within the rectangle, and other editing and display information.

e flag: One of three flag bits in the 65C816 processor that programs use to control the processor's operating modes. The setting of the e flag determines whether the processor is in native mode (6502) or emulation mode (65816). See also **m flag, x flag.**

empty handle: A handle pointing to a NIL master pointer.

emulate: To operate in a way identical to a different system. For example, the 65C816 microprocessor in the Apple IIGS can carry out all the instructions in a program originally written for an Apple II that uses a 6502 microprocessor, thus emulating the 6502.

emulation mode: The 8-bit configuration of the 65C816 processor in which it functions like a 6502 processor in all respects except clock speed.

enable: To make responsive to user manipulation. A dialog or menu that is enabled can be selected by the user. Enabling does not affect how an item is displayed. Compare **activate.**

end-of-file: See **EOF.**

EOF (end-of-file): The logical size of a ProDOS 16 file; it is the number of bytes that may be read from or written to the file.

erasing: In QuickDraw II, to color an area with the background pattern.

error: The state of a computer after it has detected a fault in one or more commands sent to it. Also called *error condition.*

error condition: See **error.**

error message: A message issued by the system or application program when it has encountered an abnormal situation or an error in data.

event: A notification to an application of some occurrence (such as an interrupt generated by a keypress) that the application may want to respond to.

event code: A numeric value assigned to each event by the Event Manager. Compare **task code.**

event-driven program: A program that responds to user inputs in real time by repeatedly testing for events. An event-driven program does nothing until it detects an event such as a click of the mouse button.

Event Manager: The Apple IIGS tool set that detects events as they happen and passes the events to the application or appropriate event handler, such as TaskMaster or GetNextEvent.

event mask: A parameter passed to an Event Manager routine to specify which types of events the routine should apply to.

event message: A field in the event record that contains additional information about the event.

event queue: A list of pending events maintained by the Event Manager.

event record: The internal representation of an event, through which your program learns all pertinent information about that event.

event type: The type of event reported to the Event Manager.

execution environment: See **operating environment.**

execution mode: One of two general states of execution of the 65C816 processor: native mode and 6502 emulation mode.

expansion slot: See **slot.**

Extended value: An 80-bit signed floating-point value with 64 bits of fraction.

extended task event record: A data structure based on the event record that contains information used and returned by TaskMaster.

FALSE: Zero. The result of a Boolean operation. The opposite of TRUE.

family name: The name identifying a font family. For example, the font family named Helvetica includes 10-point Helvetica, 12-point Helvetica Bold, and 36-point Helvetica Underlined. See also **font family.**

family number: The number identifying a font family. There is a one-to-one correspondence between family number and family name; that is, any two fonts with the same family number should have the same family name.

FamSpecBits: A bit flag in the Font Manager that restricts the range of font families available to a calling routine.

FamStatBits: A bit flag in the Font Manager that reports on the status of a font family.

file: Any named, ordered collection of information stored on a disk. Application programs and operating systems on disks are examples of files; so also are text or graphics materials created by applications and saved on disks. Text and graphics files are also called *documents*.

filename: The string of characters that identifies a particular file within its directory. ProDOS filenames may be up to 15 characters long. Compare **pathname.**

file type: An attribute of a ProDOS file that characterizes its contents and indicates how the file may be used. On disk, file types are stored as numbers; in a directory listing, they are often displayed as three-character or single-word mnemonic codes. Compare **auxiliary type.**

filling: In QuickDraw II, using a specified pattern and the drawing mask to fill the interior of a shape.

fill mode: A display option in Super Hi-Res 320 mode. In fill mode, pixels in memory with the value 0 are automatically assigned the color of the previous nonzero pixel on the scan line; the program thus need assign explicit pixel values only to *change* pixel colors.

filter procedure: A procedure that allows the application programmer to control the types of events handled by the Dialog Manager.

firmware: Programs stored permanently in ROM; most provide an interface to system hardware. Such programs (for example, the Monitor program) are built into the computer at the factory. They can be executed at any time but cannot be modified or erased. Compare **hardware, software.**

fixed: Not movable in memory once allocated. Program segments that must not be moved are placed in fixed memory blocks. Also called *immovable.* The opposite of *movable.*

fixed address: A memory block that must be at a specified address when allocated.

fixed bank: A block of memory that must start in a specified bank.

Fixed value: A 32-bit signed value with 16 bits of fraction.

flag: A variable whose value (usually 1 or 0, standing for TRUE or FALSE) indicates whether some condition holds or whether some event has occurred. A flag is used to control the program's actions at some later time.

folder: The visual representation of a subdirectory. See also **subdirectory.**

font: In typography, a complete set of type in one size and style of character. In computer usage, a collection of letters, numbers, punctuation marks, and other typographical symbols with a consistent appearance; the size and style can be changed readily. See also **font scaling.**

font bounds rectangle: The smallest rectangle that would enclose all the pixels of every character in a font; that is, the rectangle that is the union of all the character bounds rectangles of the characters in the font.

font family: All fonts that share the same name but may vary in size or style. For example, all fonts named Helvetica are in the same family, even though that family contains Helvetica, Helvetica Narrow, and Helvetica Bold.

font height: The vertical distance from a font's ascent line to its descent line.

font ID: A number that specifies a font by family, style, and size.

font ID record: A record containing the number that specifies a font by family, style, and size.

Font Manager: The Apple IIGS tool set that allows applications to use different fonts.

font rectangle: The smallest rectangle that would completely enclose all the foreground pixels of the characters of a font if the characters were drawn so that their character origins coincided.

font scaling: A process by which the Font Manager creates a font at one size by enlarging or reducing characters in an existing font of another size.

font size: The size of a font in points, from 1 to 255. The Font Manager defines the font size as a byte; QuickDraw II and the Apple IIGS font record define the font size as a word.

FontSpecBits: A bit flag in the Font Manager that restricts the range of fonts available to a calling routine.

FontStatBits: A bit flag in the Font Manager that reports on the status of a font.

font strike: A 1-bit-per-pixel pixel image consisting of the character images of every defined character in the font, placed sequentially in order of increasing ASCII code.

font style: The style in which a font was designed. The Font Manager defines the style style as a byte; QuickDraw II and the Apple IIGS font record define the font style as a word.

font substitution: An option in the LaserWriter style dialog box in the Print Manager, font substitution tells the system to substitute one font for another if the specified font is not available on the LaserWriter.

foreground color: The color of the foreground pixels in text; by default it is white.

foreground pixels: In a character image, the pixels corresponding to the character itself; that is, the bits set to 1 in the image.

FPT: See **function pointer table.**

Frac value: A 32-bit signed value with 30 bits of fraction.

fragmentation: A condition in which free (unallocated) portions of memory are scattered due to repeated allocation and deallocation of blocks by the Memory Manager.

frame region: The part of a window that surrounds the window's content region and contains standard window controls.

framing: In QuickDraw II, using the current pen size, pen pattern, drawing mask, and pen mode to draw an outline of a shape.

full native mode: See **native mode.**

full pathname: The complete name by which a file is specified, starting with the volume directory name. A full pathname always begins with a slash (/) because a volume directory name always begins with a slash. See also **pathname.**

function pointer table (FPT): A table, maintained by the Tool Locator, that points to all routines in a given tool set.

GCB: See **generator control block.**

general logic unit: See **GLU.**

generator: In the swap mode of the DOC, a functional unit formed from a pair of oscillators.

generator control block (GCB): A 16-byte block in the sound routines' work area that controls one generator.

GetNextEvent: The Event Manager call that an application can make on each cycle through its main event loop. Compare **TaskMaster.**

global coordinates: The coordinate system assigned to a pixel image (such as screen memory) that QuickDraw II draws to. In global coordinates, the boundary rectangle's origin (top left corner) has the value (0,0). Compare **local coordinates.**

GLU (general logic unit): A class of custom integrated circuits used as interfaces between different parts of the computer.

go-away region: A region in a window frame, corresponding to the close box. Clicking inside this region of the active window makes the window close or disappear.

GrafPort: A data structure (record) that specifies a complete drawing environment, including such elements as a pixel image, boundaries within which to draw, a character font, patterns for drawing and erasing, and other pen characteristics.

graphic port: A specification for how and where QuickDraw II draws. A graphic port is defined by its GrafPort record; an application may have more that one graphic port open at one time, each defined by its own GrafPort. Same as *drawing environment*.

grow image: A dotted outline of an entire window plus the lines delimiting the title bar, size box, and scroll bar areas. The image can be pulled around to follow the movements of the mouse until the mouse button is released.

grow region: A window region in which dragging changes the size of the window.

handle: See **memory handle.**

hardware: In computer terminology, the machinery that makes up a computer system. Compare **firmware, software.**

Heartbeat routines: Routines that execute at the heartbeat interrupt signal, during the vertical blanking interval (every 1/60 of a second).

hex: See **hexadecimal.**

hexadecimal, hex: The representation of numbers in the base-16 system, using the ten digits 0 through 9 and the six letters A through F. Each hexadecimal digit corresponds to a sequence of four binary digits, or bits. Hexadecimal numbers are usually preceded by a dollar sign ($).

hide: To make invisible on the screen (but not necessarily to discard).

highlight: To make something visually distinct. For example, when a button on a dialog box is selected, it appears as light letters on a dark background, rather than dark on light. An active window or control is highlighted differently than an inactive one.

horizontal blanking: The interval between the drawing of each scan line on a video display.

hot spot: The interval between the drawing of each scan line on a video display.

Human Interface Guidelines: Apple Computer's set of conventions and suggestions for writing desktop programs. Programs that follow the *Human Interface Guidelines: The Apple Desktop Interface* present a consistent and friendly interface to users.

icon: An image that graphically represents an object, a concept, or a message.

i flag: A bit in the 65C816 microprocessor's Processor Status register that disables interrupts if set to 1.

image: A representation of the contents of memory. A code image consists of machine-language instructions or data that may be loaded unchanged into memory. See also **pixel image.**

image pointer: In QuickDraw II, the pointer to the first byte of a pixel image.

image width: (1) Part of the QuickDraw II *locInfo* record that specifies the width of each line of a pixel image; the width must be an even multiple of 8 bytes. (2) For characters, same as *character image width*.

immovable: See **fixed.**

inactive: Said of controls that have no meaning or effect in the current context, such as an Open button when no document has been selected to be opened. These inactive controls are not affected by the user's mouse action and are dimmed on the screen. Compare **disable.**

index register: A register in a computer processor that holds an index for use in indexed addressing. The 6502 and 65C816 microprocessors used in the Apple II family of computers have two index registers, called the *X register* and the *Y register*.

indicator: On a dial type of control, the moving part that displays the current setting.

information bar: An optional component of a window. If present, the information bar appears just below the title bar. It may contain any application-defined information.

initialization segment: A segment in an initial load file that is loaded and executed independently of the rest of the program. It is commonly executed first, to perform any initialization that the program may require.

input/output: See **I/O.**

insertion point: The place in a document where something will be added; it is selected by clicking and is normally represented by a blinking vertical bar.

Integer value: A 16-bit signed or unsigned value.

Integer Math String: An ASCII string with no length indication supplied by the string itself.

Integer Math Tool Set: The Apple IIGS tool set that performs simple mathematical functions on integers and other fixed-point numbers and converts numbers to their ASCII string-equivalents.

interface library: A set of variable- and data-structure definitions that link a program (such as a C application) with software written in another language (such as the Apple IIGS Toolbox).

interrupt: A temporary suspension in the execution of a main program that allows the computer to perform some other task, typically in response to a signal from a peripheral device or other source external to the computer.

interrupt environment: The machine state, including register length and contents, that the interrupt handler executes within.

interrupt mode: A mode in which interrupts are used to synchronize drawing with vertical blanking.

inverting: In QuickDraw II, using the drawing mask to invert the pixels in the interior of a shape.

I/O (input/output): A general term that encompasses input/output activity, the devices that accomplish it, and the data involved.

I/O space: The portion of the memory map in a standard Apple II (and in banks $E0 and $E1 of an Apple IIGS) with addresses between $C000 and $CFFF. Programs perform I/O by writing to or reading from locations in this I/O space.

item: A component of a dialog box, such as a button, a text field, or an icon.

item descriptor: In a dialog box, a pointer or a handle to additional information concerning a dialog item.

item ID: A unique number that defines an item in a dialog box and allows further reference to it.

item line: The line of text that defines a menu item's name and appearance.

item list: A list of information about all the items in a dialog box or an alert box.

item template: A record that contains information about the items in a dialog box.

item type: Identifies the type of dialog item, usually represented by a predefined constant (such as `statText`) or a series of constants.

item value: In a dialog box, additional information concerning a dialog item.

job dialog box: A dialog box presented when the user selects Print from the File menu.

job subrecord: A field in the print record that contains information about a particular printing job. See also **print record.**

journaling mechanism: A mechanism that can separate the Event Manager from the user and feed the manager events from a file.

JSL (Jump to Subroutine Long): A 65816 assembly-language instruction that requires a long (3-byte) address. JSL can be used to transfer execution to code in another memory bank.

JSR (Jump to Subroutine): A 6502 and 65816 assembly-language instruction that requires a 2-byte address.

Jump Table: (1) A table constructed in memory by the System Loader from all Jump Table segments encountered during a load. The Jump Table contains all references to dynamic segments that may be called during execution of the program. (2) The mechanism the Sound Tool Set uses to find a low-level sound routine.

Jump Table segment: A segment in a load file that contains all references to dynamic segments that may be called during execution of that load file. The Jump Table segment is created by the linker. In memory, the loader combines all Jump Table segments it encounters into the Jump Table.

K: See **kilobyte.**

kerning: The situation that occurs when a character has foreground pixels to the left of the old pen position or to the right of the new pen position or both. When kerning occurs, the character images of adjacent characters may overlap.

keyboard equivalent: The combination of the Apple key and another key, used to invoke a menu item from the keyboard.

key-down event: An event type caused by the user pressing any character key on the keyboard or keypad. The character keys include all keys except Shift, Caps Lock, Control, Option, and Apple, which are called *modifier keys*. Modifier keys are treated differently and generate no keyboard events of their own.

kilobyte (K): A unit of measurement consisting of 1024 (2^{10}) bytes. In this usage, *kilo* (from the Greek, meaning a thousand) stands for 1024. Thus, 64K memory equals 65,536 bytes. See also **megabyte.**

landscape mode: A printing mode in which text is printed top to bottom (that is, longways) on the paper.

leading: Pronounced "LED-ing." The space between lines of text. It is the number of pixels vertically between the descent line of one character and the ascent line of the character immediately beneath it.

leftward kern: For characters, the distance in pixels from the character origin to the left edge of the character.

length byte: The first byte of a Pascal string. It specifies the length of the string, in bytes.

library, library file: An object file containing program segments, each of which can be used in any number of programs. The linker can search through the library file for segments that have been referenced in the program source file.

library file: See **library.**

limit rectangle: The rectangle that limits the travel of a region that is being dragged with the mouse.

line: In QuickDraw II, an infinitely thin graphic object that is represented by its ends, which are defined by two points.

LineEdit Tool Set: The Apple IIGS tool set that provides simple text-editing functions. It is used mostly in dialog boxes.

line height: The total amount of vertical space from line to line in a text document. Line height is the sum of ascent, descent, and leading.

list: As defined by the List Manager, a scrollable, vertical arrangement of similar items on the screen; the items are selectable by the user.

list control: A custom control created by the List Manager.

list control record: A data structure that defines the appearance of a list control after the control has been created.

list record: A data structure that defines the initial appearance of a list control.

List Manager: The Apple IIGS Tool set that allows an application to present the user with a list from which to choose (for example, the Font Manager uses the List Manager to arrange lists of fonts).

local coordinates: A coordinate system unique to each GrafPort and independent of the global coordinates of the pixel image that the port is associated with. For example, local coordinates do not change as a window is dragged across the screen; global coordinates do not change as a window's contents are scrolled.

location table: In a font, an array of integers with an entry for each character code.

locInfo: Acronym for *location information*. The data structure (record) that ties the coordinate plane to an indvidual pixel image in memory.

lock: To prevent a memory block from being moved or purged. A block may be locked or unlocked by a call to the Memory Manager. Compare **unlock.**

long, long word: On the Apple IIGS, a 32-bit (4-byte) data type.

Longint value: A 32-bit signed or unsigned value.

Macintosh: A family of Apple computers; for example, the Macintosh 512K and the Macintosh Plus. Macintosh computers have high-resolution screens and use mouse devices for choosing commands and for drawing pictures.

macro: A single keystroke or command that a program replaces with several keystrokes or commands. For example, the APW Editor allows you to define macros that execute several editor keystroke commands; the APW Assembler allows you to define macros that execute instructions and directives. Macros are almost like higher–level language instructions, making assembly-language programs easier to write and complex keystrokes easier to execute.

macro library: A file of related macros.

main event loop: The central routine of an event-driven program. During execution, the program continually cycles through the main event loop, branching off to handle events as they occur and then returning to the event loop.

mainID: A subfield of the user ID. Each running program is assigned a unique *mainID* field.

manager: See **tool set.**

mask: A parameter, typically one or more bytes long, whose individual bits are used to set or block particular features. See, for example, **event mask.**

master color value: A 2-byte number that specifies the relative intensities of the red, green, and blue signals output by the Apple IIGS video hardware.

master pointer: A fixed location that always contains the address of a specified block, regardless of whether the block is moved. See also **memory handle.**

master user ID: The value of a user ID *disregarding the contents of the auxID field.* If an application allocates various memory blocks and assigns them unique IDs consisting of *auxID* values added to its own user ID, then all will share the same master user ID and all can be purged or disposed with a single call.

Mb: See **megabyte.**

megabyte (Mb): A unit of computer memory or disk drive capacity equaling 1,048,576 bytes.

megahertz (MHz): A unit of measurement of frequency, equal to 1,000,000 hertz (cycles per second).

memory attributes word: A word that determines how a specified memory block is allocated and maintained.

memory block: See **block** (2).

memory expansion card: A memory card that increases Apple IIGS internal memory capacity beyond 256K, up to 8 megabytes

memory fragmentation: See **fragmentation.**

memory handle: A number that identifies a memory block. A handle is a pointer to a pointer; it is the address of a master pointer, which in turn contains the address of the block. Also called simply **handle.**

Memory Manager: The Apple IIGS Tool set that manages memory use. The Memory Manager keeps track of how much memory is available and allocates memory blocks to hold program segments or data.

menu: A list of choices presented by a program, from which the user can select an action. See also **pull-down menu.**

menu bar: The horizontal strip at the top of the screen that contains menu titles for the pull-down menus.

menu bar record: A data structure that contains the menu position, color, menu lists, item lists, and other flags the Menu Manager needs to manage menus.

menu definition procedure: A procedure used to define the appearance and behavior of a custom menu.

menu ID: A number in the menu record that identifies an individual menu.

menu item: On a menu, the text of a command or a line dividing groups of choices.

menu line: A line of text plus code characters that defines the appearance of a particular menu title.

Menu Manager: The Apple IIGS Tool Set that maintains the pull-down menus and the items in the menus.

menu record: A data structure that provides information about one of the menus in a menu bar.

m flag: One of three flags in the 65816 microprocessor's Processor Status register that controls execution mode. When the m flag is set to 1, the accumulator is 8 bits wide; otherwise, it is 16 bits wide. See also **e flag, x flag.**

MHz: See **megahertz.**

microprocessor: A central processing unit that is contained in a single integrated circuit. The Apple IIGS uses a 65C816 microprocessor.

minimum blink interval: The minimum time between blinks of the caret.

minimum version number: The minimum version of a particular tool set that an application needs to function.

minipalette: In Super Hi-Res 640 mode, a quarter of the color table. Each pixel in 640 mode can have one of four colors specified in a minipalette.

Miscellaneous Tool Set: The Apple IIGS tool set that includes mostly system-level routines that must be available for other tool sets.

missing character: In a font, a character that does not have a defined symbol.

missing symbol: In a font, the symbol substituted for any ASCII value for which the font does not have a defined symbol. In the Apple IIGS system font, the missing symbol is a box containing a question mark.

modal dialog box: A dialog box that puts the machine in a state where the user cannot execute functions outside of the dialog box until the dialog box is closed. Compare **modeless dialog box.**

mode: A state of a computer or system that determines its behavior. A manner of operating.

modeless dialog box: A dialog box that does not require the user to respond before doing anything else. Unlike a modal dialog box, it is possible to keep working even if the box is still in use. Compare **modal dialog box.**

modifier keys: The Shift, Caps Lock, Control, Option, and Apple keys. Such keys do not generate a keyboard event by themselves; rather, their states are recorded whenever another event is posted.

monospaced: Said of a font whose character widths are all identical. Compare **proportionally spaced.**

mouse: A small device the user moves around on a flat surface next to the computer. The mouse controls a pointer on the screen whose movements correspond to those of the mouse. The pointer selects operations, moves data, and draws graphic objects.

mouse button: A button on a mouse device with which the user selects objects on the screen.

mouse clamps: Values that establish the minimum and maximum X and Y coordinates for the mouse.

mouse-down: An action or an event, signifying that the user has pressed the mouse button.

mouse-up: An action or an event, signifying that the user has released the mouse button.

movable: Able to be moved to different memory locations during program execution (a memory block attribute). The opposite of *fixed*.

native mode: The 16-bit operating configuration of the 65C816 microprocessor.

NDA: See **new desk accessory.**

new desk accessory (NDA): A desk accessory designed to execute in a desktop, event-driven environment. Compare **classic desk accessory.**

NIL: A value of 0. A pointer is NIL if its value is all zeros. A memory handle is NIL if the address it points to is filled with zeros. Handles to purged memory blocks are NIL.

nonspecial, normal memory: Memory that has no special restrictions on it. On the Apple IIGS, such memory includes banks \$2–\$DF and parts of banks \$E0 and \$E1.

Note Sequencer: The Apple IIGS tool set that makes it possible to play music asynchronously in programs.

Note Synthesizer: An Apple IIGS tool set that facilitates creation and manipulation of musical notes.

null event: An event reported when there are no other events to report.

object module format (OMF): The file format used in Apple IIGS object files, library files, and load files. Compare **text file format.**

offset: The number of character positions or memory locations away from a point of reference.

OK: One of two predefined item ID numbers for dialog box buttons (OK = 1). Compare **Cancel.**

OMF: See **object module format.**

operating environment: The overall hardware and software setting within which a program runs. Also called *execution environment.*

operating system: A general-purpose program that organizes the actions of the various parts of the computer and its peripheral devices. See also **disk operating system.**

origin: (1) The first memory address of a program or of a portion of one. The first instruction to be executed. (2) The location (0,0) on the QuickDraw II coordinate plane, in either global coordinates or local coordinates. (3) The top left corner of any rectangle (such as a boundary rectangle or a port rectangle) in QuickDraw II. (4) See **character origin.**

oscillator: A device that generates a vibration. In the Apple IIGS digital oscillator chip, an oscillator is an address generator that points to the next data byte in memory that represents part of a particular sound wave.

oval: A circle or an ellipse, one of the fundamental classes of objects drawn by QuickDraw II.

pack: To compress data into a smaller space to conserve storage space.

page: (1) A portion of memory 256 bytes long and beginning at an address that is an even multiple of 256. Memory blocks whose starting addresses are an even multiple of 256 are said to be *page aligned.* (2) An area of main memory containing text or graphic information being displayed on the screen.

page-aligned: Said of a memory block that starts at a memory address that is an even multiple of 256 (a memory block attribute). See also **page** (1).

paging region: In a scroll bar, the area a user clicks to move the view of the data a page at a time.

painting: In QuickDraw II, using the current pen pattern, drawing mask, and pen mode to fill the interior of a shape.

palette: The full set of colors available for an individual screen pixel.

parameter: A value passed to or from a function or other routine.

parameter RAM: RAM on the Apple IIGS clock chip. A battery preserves the clock settings and the RAM contents when the power is off. Control Panel settings are kept in battery RAM.

part code: A number between 1 and 255 that stands for a particular part of a control. The Control Manager uses part codes to allow different parts of a control to respond in different ways.

partial pathname: A pathname that includes the filename of the desired file but excludes the volume directory name (and possibly one or more of the subdirectories in the path). It is the part of a pathname following a prefix; a prefix and a partial pathname together constitute a full pathname. A partial pathname does not begin with a slash because it has no volume directory name.

Pascal: A high-level programming language. Named for the philosopher and mathematician Blaise Pascal.

Pascal string: An ASCII character string preceded by a single byte whose numerical value is the number of characters in the string. Compare **C string.**

Pascal-type string: Same as *Pascal string*.

paste: To place the desk scrap (contents of the Clipboard—whatever was last cut or copied) at the insertion point.

pathname: A name that specifies a file. It is a sequence of one or more filenames separated by slashes, tracing the path through subdirectories that a program must follow to locate the file. See also **full pathname, partial pathname, prefix.**

pathname prefix: See **prefix.**

pattern: An 8-by-8 pixel image, used to define a repeating design (such as stripes) or color.

PB register: See **program bank register.**

PC register: A register within the 65816 microprocessor that keeps track of the memory address of the next instruction to be executed. PC stands for *program counter*.

pen: The conceptual tool with which QuickDraw II draws shapes and characters. Each GrafPort has its own pen.

pen location: The position (on the coordinate plane) at which the next character or line will be drawn.

pen mode: One of several Boolean operations that determine how the pen pattern is to affect an existing image. Compare **text mode.**

pen pattern: See **pattern.**

pen size: The size of the rectangle that will be used as the drawing pen.

peripheral card: A hardware device placed inside a computer and connected to one of the computer's peripheral expansion slots. Peripheral cards perform a variety of functions, from controlling a disk drive to providing a clock/calendar.

peripheral device: See **device.**

picture: A saved sequence of QuickDraw II drawing commands (and, optionally, picture comments) that you can play back later with a single procedure call; also, the image resulting from these commands.

pinning: The process of assigning positive overflows to the largest positive representable value and negative overflows to the largest negative representable value.

pixel: A contraction of *picture element,* the smallest dot you can draw on the screen. Also a location in video memory that corresponds to a point on the graphics screen when the viewing window includes that location. In the Super Hi-Res display on the Apple IIGS, each pixel is represented by either 2 or 4 bits. See also **pixel image.**

pixel image: A graphics image picture consisting of a rectangular grid of pixels.

plain-styled: Said of a font or character that is not bold, italicized, underlined, or otherwise styled apart from ordinary text.

plane: The front-to-back position of a window on the desktop.

point: (1) A unit of measurement for type; 12 points equal 1 pica, and 6 picas equal 1 inch; thus, 1 point equals 1/72 inch. (2) A relative measure (taken from the type measure) used to distinguish font size on output devices. (3) In QuickDraw II, the Y and X coordinates of a location on the coordinate plane.

pointer: An item of information consisting of the memory address of some other item. For example, the 65C816 stack register contains a pointer to the top of the stack.

pointing device: Any device, such as a mouse, graphics tablet, or light pen, that can be used to specify locations on the computer screen.

polygon: Any sequence of connected lines.

port: (1) A socket on the back panel of the computer where the user can plug in a cable to connect a peripheral device, another computer, or a network. (2) A graphic port (GrafPort).

portrait mode: A printing mode in which text prints from left to right on the paper.

port rectangle: A rectangle that describes the active region of a GrafPort's pixel map—the part that QuickDraw II can draw into. The content region of a window on the desktop corresponds to the window's port rectangle.

portRect: The GrafPort field that defines the port's port rectangle.

post: To place an event in the event queue for later processing.

prefix: A pathname starting with a volume name and ending with a subdirectory name. It is the part of a full pathname that precedes a partial pathname; a prefix and a partial pathname together constitute a full pathname. A prefix always starts with a slash (/) because a volume directory name always starts with a slash.

prestyled: Said of a font that has a certain style or combination of styles built into the font's design.

printer information subrecord: A data structure within the print record that contains the information needed for page composition.

printing loop: The page-by-page cycle that an application goes through when it prints a document.

Print Manager: The Apple IIGS tool set that allows an application to use standard QuickDraw II routines to print text or graphics on a printer.

print record: A record containing all the information needed by the Print Manager to perform a particular printing job.

private scrap: A buffer (and its contents) set up by an application for cutting and pasting, analogous to but apart from the desk scrap.

ProDOS: Acronym for *Professional Disk Operating System.* A family of disk operating systems developed for the Apple II family of computers. It includes ProDOS 8 and ProDOS 16.

ProDOS 8: A disk operating system developed for standard Apple II computers. It runs on 6502-series microprocessors and on the Apple IIGS when the 65C816 processor is in 6502 emulation mode.

ProDOS 16: A disk operating system developed for 65C816 native-mode operation on the Apple IIGS. It is functionally similar to ProDOS 8 but more powerful.

program bank register: The 65C816 register whose contents form the high-order byte of all 3-byte code address operands. Also called *PB register*.

program counter: See **PC register.**

program status register: A register in the 65C816 microprocessor that contains flags reflecting the various aspects of machine state and operation results.

proportionally spaced: Said of a font whose characters vary in width, so the amount of horizontal space needed for each character is proportional to its width. Compare **monospaced.**

pseudo-type: A type that provides some additional information about a parameter of a toolbox routine.

pull-down menu: A set of choices for actions that appears near the top of the display screen in a desktop application, usually overlaying the present contents of the screen without disrupting them. Dragging through the menu and releasing the mouse button while a command is highlighted chooses that command.

purge: To temporarily deallocate a memory block. The Memory Manager purges a block by setting its master pointer to NIL (0). All handles to the pointer are still valid, so the block can be reconstructed quickly. Compare **dispose.**

purge level: A memory block attribute, indicating that the Memory Manager may purge the block if it needs additional memory space. Purgeable blocks have different purge levels, or priorities for purging; these levels are set by Memory Manager calls.

queue: A list in which entries are added (pushed) at one end and removed (pulled) at the other end, causing entries to be removed in first-in, first-out (FIFO) order. Compare **stack.**

QuickDraw II: The Apple IIGS tool set that controls the graphics environment and draws simple objects and text. Other tools call QuickDraw II to draw such things as windows.

QuickDraw II Auxiliary: The Apple IIGS tool set that provides extensions to the capabilities of QuickDraw II.

QuickDraw II Auxiliary icon record: A data structure that defines the appearance of an icon.

quit: To terminate execution in an orderly manner. Apple IIGS applications quit by making a ProDOS 16 QUIT call or the equivalent.

radio button: A common type of control in dialog boxes. Radio buttons are small circles organized into families; clicking any button on turns off all the others in the family, like the buttons on a car radio. See also **check box.**

RAM: See **random-access memory.**

random-access memory (RAM): Memory in which information can be referred to in an arbitrary or random order. Programs and other data in RAM are lost when the computer is turned off. Technically, the *read-only memory* (ROM) is also random access, and what's called RAM should correctly be termed *read-write memory*. Compare **read-only memory.**

range mode: In the List Manager, a selection mode that allows the user to select a range of members in a list.

read-only memory (ROM): Memory whose contents can be read, but not changed; used for storing firmware. Information is placed into ROM once, during manufacture; it then remains there permanently, even when the computer's power is turned off. Compare **random-access memory.**

real font: A font that exists on disk or was added by an application and marked as *real*. Compare **unreal font.**

rectangle: One of the fundamental shapes drawn by QuickDraw II. Rectangles are completely defined by two points—their upper left and lower right corners on the coordinate plane. The upper left corner of any rectangle is its origin.

reentrant: Said of a routine that is able to accept a call while one or more previous calls to it are pending, without invalidating the previous calls. Under certain conditions, the Apple IIGS Scheduler manages execution of routines that are not reentrant.

region: An arbitrary area or set of areas on the QuickDraw II coordinate plane. The outline of a region must be one or more closed loops.

relocatable: Characteristic of a load segment or other program code that includes no references to specific address, and so can be loaded at any memory address. A relocatable segment consists of a code image followed by a relocation dictionary. Compare **absolute.**

relocation: The act of modifying a program in memory so that its address operands correctly reflect its location and the locations of other segments in memory. Relocation is performed by the system loader when a relocatable segment is first loaded into memory.

repeat delay: The time interval before the first auto-key event is generated.

repeat speed: The time interval between auto-key events, except for the first auto-key event. See also **repeat delay.**

reserved memory: Memory not managed by the Memory Manager; that is, memory that is marked as busy at startup time.

right scroll bar: The control the user selects to scroll vertically through the data in the window.

ROM: See **read-only-memory.**

ROM font: The font contained in system ROM.

rounded result: The nearest representable value to the actual value, with ties going to the value with the larger magnitude.

rounded-corner rectangle: One of the fundamental shapes drawn by QuickDraw II. The rounded corners of this type of rectangle are defined by an oval height and oval width.

routine: A part of a program that accomplishes some task subordinate to the overall task of the program.

RTL (Return from Subroutine Long): A 65816 assembly-language instruction.

RTS (Return from Subroutine): A 65816 assembly-language instruction.

SANE (Standard Apple Numeric Environment): The set of methods that provides the basis for floating-point calculations in Apple computers. SANE meets all requirements for extended-precision, floating-point arithmetic as prescribed by IEEE Standard 754 and ensures that all floating-point operations are performed consistently and return the most accurate results possible.

SANE Tool Set: The Apple IIGS tool set that performs high-precision, floating-point calculations, following SANE standards.

scaled font: A font that is created by the Font Manager by calculation from a real font of a different size.

scaling: The process of taking all characters of a real font and making them bigger or smaller to generate a requested font.

scan line: A single horizontal line of pixels on the screen. It corresponds to a single sweep of the electron gun in the video display tube.

scan line control byte (SCB): A byte in memory that controls certain properties, such as available colors and number of pixels, for a scan line on the Apple IIGS. Each scan line has its own SCB.

SCB: See **scan line control byte.**

Scheduler: The Apple IIGS tool set that manages requests to execute interrupted software that is not reentrant. If, for example, an interrupt handler needs to make system software calls, it must do so through the Scheduler because ProDOS 16 is not reentrant. Applications normally need not use the Scheduler because ProDOS 16 is not in an interrupted state when it processes applications' system calls.

scrap count: A count that indicates how many times the desk scrap has changed.

Scrap Manager: The Apple IIGS tool set that supports the desk scrap, which allows data to be copied from one application to another or from one place to another within an application.

scroll: To move an image of a document or directory in its window so that a different part of it is visible.

scroll bar: A rectangular bar that may be along the right or bottom of a window. Clicking or dragging in the scroll bar causes the view of the document to change.

selection range: The series of characters where the next editing action will occur.

serial interface: A standard method, such as RS-232, for transmitting data serially (as a sequence of bits).

serial port: The connector for a peripheral device that uses a serial interface.

Shaston: The Apple IIGS system font.

shut down: To remove from memory or otherwise make unavailable, as a tool set that is no longer needed or an application that has quit.

single mode: In the List Manager, a selection mode that allows the user to select only one member of a list at once; that is, when the user drags the mouse, the selection moves from one member to another.

65C816: The microprocessor used in the Apple IIGS. The 65C816 is a CMOS device with a 16-bit data bus and a 24-bit address bus.

6502: The microprocessor used in the Apple II, Apple II Plus, and early models of the Apple IIe. The 6502 is an NMOS device with 8-bit data registers and 16-bit address registers.

65816 assembly language: A low-level programming language written for the 65816 family of microprocessors.

640 mode: An Apple IIGS video display mode, 640 pixels horizontally by 200 pixels vertically.

size box: A small region in the lower right corner of a window that the user can drag to change the size of the window.

slop rectangle: The rectangle that allows the user some margin for error when moving the mouse.

slot: A narrow socket inside the computer where the user can install peripheral cards. Also called *expansion slot.*

smoothing: A LaserWriter printing option that asks the system to smooth out any bit-mapped fonts with jagged edges.

software: A collective term for programs, the instructions that tell the computer what to do. Software is usually stored on disks. Compare **firmware, hardware.**

sound GLU (general logic unit): The interface chip between the system hardware and the sound hardware.

Sound Tool Set: The Apple IIGS tool set that provides low-level access to the sound hardware.

source location: The location (memory buffer or portion of the QuickDraw II coordinate plane) from which data such as text or graphics are copied. See also **destination rectangle.**

source rectangle: The rectangle (on the QuickDraw II coordinate plane) where text or graphics are drawn when transferred from somewhere else. Compare **destination rectangle.**

special memory: On an Apple IIGS, all of banks $00 and $01 and all display memory in banks $E0 and $E1. It is the memory directly accessed by standard Apple II programs running on the Apple IIGS.

spool printing: A two-step printing method used to print graphics on the ImageWriter. In the first step, it writes out (spools) a representation of your document's printed image to a disk file or to memory. The second step consists of this information being converted into a bit image and printed. Compare **draft printing.**

S register: See **stack register.**

SRQ list: A special tool mechanism that can be used to poll the Apple Desktop Bus for data from specific devices.

SRQ list completion routine: Used in conjunction with the ADB Tool Set routine AsyncADBReceive, this completion routine obtains ADB data from a buffer. The only major difference between this routine and the AsyncADBReceive completion routine is that the SRQ list routine has an extra return address on the stack when it is called. Compare **AsyncADBReceive completion routine.**

stack: A list in which entries are added (pushed) and removed (pulled) at one end only (the top of the stack), causing entries to be removed in last-in, first-out (LIFO) order. *The stack* usually refers to the particular stack pointed to by the 65C816's stack register. Compare **queue.**

stack pointer: See **stack register.**

stack register: A register in the 65816 processor that indicates the next available memory address in the stack. Also called *S register.*

stage byte: Determines the actions taken by an alert. See also **alert stage.**

Standard Apple Numerics Environment: See **SANE.**

standard Apple II: Any computer in the Apple II family except the Apple IIGS. That includes the original Apple II, the Apple II Plus, the Apple IIe, and the Apple IIc. Compare **Apple II.**

Standard File Operations Tool Set: The Apple IIGS tool set that creates a standard user interface for opening and closing files.

standard window controls: The window controls that allow the user to scroll through the data in the window, change the window's shape, or close the window. They also provide information about the document currently displayed in the window.

start up: To get the system, application program, or tool set running.

static segment: A program segment that must be loaded when the program is started and cannot be removed from memory until execution terminates. Compare **dynamic segment.**

static text: Text on the screen that cannot be altered by the user.

string: A sequence of characters. See also **C string, Pascal string.**

string bounds rectangle: The smallest rectangle that would enclose all the foreground and background pixels of a string if the string were to be drawn.

structure region: An entire window: its content region plus its frame region.

style dialog box: A dialog box that allows the user to specify formatting information, page size, and printer options.

styled variation: An italicized, bold, underlined, or otherwise altered version of a plain-styled character or font.

style subrecord: A data structure within the print record that contains information gathered from the user via the style and job dialog boxes.

subdirectory: A file that contains information about other files. In a hierarchical file system, files are accessed through the subdirectories that reference them.

subroutine: A part of a program that can be executed on request from another point in the program and that, upon completion, returns control to the point of the request.

Super Hi-Res: Either of two high-resolution Apple IIGS display modes: 320 mode consists of an array of pixels 320 wide by 200 high, with 16 available colors; 640 mode is an array 640 wide by 200 high, with 16 available colors (with restrictions).

swap pair: A pair of oscillators that form a functional unit (called a *generator*) when the digital oscillator chip (DOC) is in swap mode.

switcher: A program that rapidly transfers execution among several applications.

switch event: An event that indicates the application is being returned to after being switched out of by a switcher-type application.

synthesizer: (1) A hardware device capable of creating sound digitally and converting it into an analog waveform that you can hear. (2) By analogy, any sound-making entity, such as the Free-Form Synthesizer in the Sound Tool Set.

system clock: See **clock** (1).

system disk: A disk that contains the operating system and other system software needed to run applications.

system event mask: A set of flags that controls which event types get posted into the event queue by the Event Manager.

system failure: The unintentional termination of program execution due to a severe software error.

System Failure Manager: A part of the Miscellaneous Tool Set that processes fatal errors by displaying a message on the screen and halting execution.

system folder: The SYSTEM/subdirectory on a ProDOS 16 system disk.

system font: The font that QuickDraw II uses as the default current font when a new GrafPort is opened.

System Loader: The program that manages the loading and relocation of load segments (programs) into the Apple IIGS memory. The System Loader works closely with ProDOS 16 and the Memory Manager.

system menu bar: The menu bar that always appears at the top of the screen in desktop applications. It contains all of the commonly used functions such as File, Edit, and so on. Compare **window menu bar.**

system software: The components of a computer system that support application programs by managing system resources such as memory and I/O devices.

system window: A window in which a desk accessory is displayed.

task code: A numeric value assigned to the result of each event handled by TaskMaster. Compare **event code.**

TaskMaster: A Window Manager routine that handles many typical events for an application. Applications may call TaskMaster instead of GetNextEvent.

template: A data structure or set of parameters that defines the characteristics of a desktop feature, such as a window control. The NewWindow parameter list is a template that defines the appearance of a window to be opened by the NewWindow call.

text block: A number of ASCII characters in a buffer, with the number specified separately.

text buffer: A 1-bit-per-pixel pixel image reserved for the private use of the QuickDraw II text-drawing call.

text file format (TFF): A file that consists of ASCII representations of characters. Compare **object module format.**

text mode: One of eight possible interactions between pixels in text being drawn to the screen and pixels on the screen that fall under characters being drawn. Compare **pen mode.**

Text Tool Set: The Apple IIGS tool set that provides an interface between Apple II character device drivers and applications running in native mode.

TFF: See **text file format.**

320 mode: An Apple IIGS video display mode, 320 pixels horizontally by 200 pixels vertically.

tick count: The (approximate) number of 60th-second intervals since system startup.

title bar: The horizontal bar at the top of a window that shows the name of the window's contents. The user can move the window by dragging the title bar.

tool: See **tool set.**

toolbox: See **Apple IIGS Toolbox.**

Tool Locator: The Apple IIGS tool set that dispatches tool calls. The Tool Locator knows and retrieves the appropriate routine when your application makes a tool call.

tool pointer table (TPT): A table, maintained by the Tool Locator, that contains pointers to all active tool sets.

tool set: A group of related routines (usually in ROM) that perform necessary functions or provide programming convenience. They are available to applications and system software. The Memory Manager, the System Loader, and QuickDraw II are Apple IIGS tool sets.

tool table: A list of all needed tool sets and their minimum required versions. An application constructs this table in order to load its tools with the LoadTools call.

TPT: See **tool pointer table.**

transfer mode: A specification of which Boolean operation QuickDraw II should perform when drawing. See, for example, **XOR.**

TRUE: Not zero. The result of a Boolean operation. The opposite of FALSE.

typeID: A subfield of the user ID. The User ID Manager assigns a *typeID* value based on the type of program (application, tool set, and so on) requesting the memory.

unclaimed interrupt: This occurs when the hardware Interrupt Request Line is active, indicating that an interrupt-producing device needs attention, but none of the installed interrupt handlers claims responsibility for the interrupt.

underline: (1) A style of text. (2) A method used to separate groups of items in a menu. An underlined item does not use any more space, on the screen or in memory, than the item does without the underline. Compare **dividing line.**

unhighlight: To restore to normal display. Selected controls, menu items, or other objects may be highlighted (usually displayed in inverse colors) while in use and unhighlighted when not in use.

unload: To remove a load segment from memory. To unload a segment, the System Loader does not actually "unload" anything; it calls the Memory Manager to either purge or dispose of the memory block in which the code segment resides.

unlock: To permit the Memory Manager to move or purge a memory block if needed. Compare **lock.**

unmovable: See **fixed.**

unpack: To restore to normal format from a packed format.

unpurgeable: Having a purge level of zero. The Memory Manager is not permitted to purge memory blocks whose purge level is zero.

unreal font: A font that was scaled by the Font Manager from a real font of a different size or added by an application and marked as *unreal.* Compare **real font.**

update event: An event posted by the Window Manager when all or part of a window needs to be redrawn.

user ID: An identification number that specifies the owner of every memory block allocated by the Memory Manager. User ID's are assigned by the User ID Manager.

User ID Manager: A part of the Miscellaneous Tool Set that is responsible for assigning user ID's to every block of memory allocated by the Memory Manager.

vector: A location that contains a value used to find the entry point address of a subroutine.

view rectangle: The rectangle within which text in an edit record is visible; that is, the portion of the text in the destination rectangle that the user can see is determined by the view rectangle.

visible region: The part of a window that's actually visible on the screen. The visible region is defined by a GrafPort field manipulated by the Window Manager.

voice: Any one of 16 pairs of oscillators in the Ensoniq sound chip on the Apple IIGS.

wedge: A filled arc, one of the fundamental shapes drawn by QuickDraw II.

window: A rectangular area that displays information on a desktop. You view a document through a window. You can open or close a window, move it around on the desktop, and sometimes change its size, scroll through it, and edit its contents. The area inside the window's frame corresponds to the port rectangle of the window's GrafPort.

window definition procedure: A procedure used to define the appearance and behavior of a custom window.

window frame: The outline of the entire window plus certain standard window controls.

Window Manager: The Apple IIGS Tool Set that updates and maintains windows.

window menu bar: A menu bar that appears at the top of the active window, below the system menu bar. It can contain document titles, applications, and functions. Compare **system menu bar.**

window record: The internal representation of a window, where the Window Manager stores all the information it needs for its operations on that window.

word: On the Apple IIGS, a 16-bit (2-byte) data type. Compare **long, long word.**

x flag: One of three flag bits in the 65C816 processor that programs use to control the processor's operating modes. In native mode, the setting of the x flag determines whether the index registers are 8 bits wide or 16 bits wide. See also **e flag, m flag.**

XOR: Exclusive-OR. A Boolean operation in which the result is TRUE if, and only if, the two items being compared are unequal in value.

X register: One of the two index registers in the 65816 microprocessor.

Y register: One of the two index registers in the 65816 microprocessor.

zero page: The first page (256 bytes) of memory in a standard Apple II computer (or in the Apple IIGS when running a standard Apple II program). Because the high-order byte of any address in this part of memory is zero, only a single byte is needed to specify a zero-page address. Compare **direct page.**

z flag: A bit in the 65816 processor's Processor Status register that is set to 1 if the last operation resulted in 0 (zero).

zoom box: A small box with a smaller box enclosed in it found on the right side of the title bar of some windows. Clicking the zoom box expands the window to its maximum size; clicking it again returns the window to its original size.

zoom region: The window region that corresponds to the zoom box.

Index

F

G

H

J

K

L

M

P

U

V

W

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using the Apple Macintosh® computers and Microsoft® Word. Proof and final pages were created on the Apple LaserWriter® Plus. POSTSCRIPT®, the LaserWriter page-description language, was developed by Adobe Systems Incorporated. Some of the illustrations were created using Adobe Illustrator™.

Text type is ITC Garamond® (a downloadable font distributed by Adobe Systems). Display type is ITC Avant Garde Gothic®. Bullets are ITC Zapf Dingbats®. Program listings are set in Apple Courier, a monospaced font.