CHAPTER 3

QuickDraw Drawing

Contents

About QuickDraw Drawing 3-3
- The Graphics Pen 3-4
- Bit Patterns 3-5
- Boolean Transfer Modes With 1-Bit Pixels 3-8
- Lines and Shapes 3-11
  - Defining Lines and Shapes 3-11
  - Framing Shapes 3-12
  - Painting and Filling Shapes 3-12
  - Erasing Shapes 3-12
  - Inverting Shapes 3-13
  - Other Graphic Entities 3-13
- The Eight Basic QuickDraw Colors 3-14

Drawing With QuickDraw 3-16
- Drawing Lines 3-17
- Drawing Rectangles 3-22
- Drawing Ovals, Arcs, and Wedges 3-25
- Drawing Regions and Polygons 3-27
- Performing Calculations and Other Manipulations of Shapes 3-31
- Copying Bits Between Graphics Ports 3-32
- Customizing QuickDraw's Low-Level Routines 3-35

QuickDraw Drawing Reference 3-36
- Data Structures 3-36
- Routines 3-41
  - Managing the Graphics Pen 3-41
  - Changing the Background Bit Pattern 3-48
  - Drawing Lines 3-49
  - Creating and Managing Rectangles 3-52
  - Drawing Rectangles 3-58
  - Drawing Rounded Rectangles 3-63
  - Drawing Ovals 3-68

Drawing Arcs and Wedges 3-71
Creating and Managing Polygons 3-78
Drawing Polygons 3-81
Creating and Managing Regions 3-85
Drawing Regions 3-100
Scaling and Mapping Points, Rectangles, Polygons, and Regions 3-104
Calculating Black-and-White Fills 3-108
Copying Images 3-112
Drawing With the Eight-Color System 3-122
Determining Whether QuickDraw Has Finished Drawing 3-125
Getting Pattern Resources 3-126
Customizing QuickDraw Operations 3-129
Resources 3-140
The Pattern Resource 3-140
The Pattern List Resource 3-141
Summary of QuickDraw Drawing 3-142
Pascal Summary 3-142
Constants 3-142
Data Types 3-144
Routines 3-145
C Summary 3-149
Constants 3-149
Data Types 3-151
Functions 3-152
Assembly-Language Summary 3-157
Data Structures 3-157
Global Variables 3-158

This chapter describes routines common to both basic QuickDraw and Color QuickDraw that you can use to draw lines, rectangles, rounded rectangles, ovals, arcs, wedges, polygons, and regions, and to copy images from one graphics port to another. This chapter also describes the routines that you can use to perform calculations and other manipulations of these shapes—including comparing them and finding their unions and intersections, and moving, shrinking, and expanding them.

Read this chapter to learn how to draw on all models of Macintosh computers. All of the routines described in this chapter depend on your application to create a graphics port drawing environment as described in the previous chapter, "Basic QuickDraw." As noted in this chapter, many of these routines have additional capabilities when performed in the more sophisticated color drawing environments described in the next chapter in this book, "Color QuickDraw." However, if your application does not use color, or uses only a few colors, you may find it unnecessary to create the drawing environment described in the chapter "Color QuickDraw."

This chapter also describes how to copy a bit image from one onscreen graphics port to another onscreen graphics port. To prevent the choppiness that can occur when you build complex images onscreen, your application should use the drawing routines described in this chapter to create complex images in offscreen graphics worlds. Your application can then copy these images to onscreen graphics ports, as described in the chapter "Offscreen Graphics Worlds" in this book.

QuickDraw also supports the creation and manipulation of pictures and text. The chapter "Pictures" in this book describes the routines for drawing pictures. For information about drawing text, see the chapter "QuickDraw Text" in *Inside Macintosh: Text*.

This chapter describes how to

- use the graphics pen
- create, draw, and manipulate the shapes supported by QuickDraw
- draw a copy of a bit image from one graphics port into another graphics port
- use the eight-color system supported by basic QuickDraw
- customize QuickDraw's drawing operations

About QuickDraw Drawing

QuickDraw provides your application with routines for rapidly creating, manipulating, and drawing graphic objects such as lines, arcs, rectangles, ovals, regions, and bitmaps.

These routines extract information from and affect the fields of the current graphics port, without specifically naming it as a parameter. For example, the `Move` procedure moves the graphics pen of the current graphics port, changing the value of its `pnLoc` field, and the `PaintOval` procedure paints an oval using the pattern and pattern mode of the graphics pen for the current graphics port.

The previous chapter, “Basic QuickDraw,” describes the basic graphics port. The next chapter, “Color QuickDraw,” describes the color graphics port. The routines described in this chapter operate in both types of graphics ports.

Whenever you use QuickDraw, all drawing is performed with the graphics pen, which is described next.

The Graphics Pen

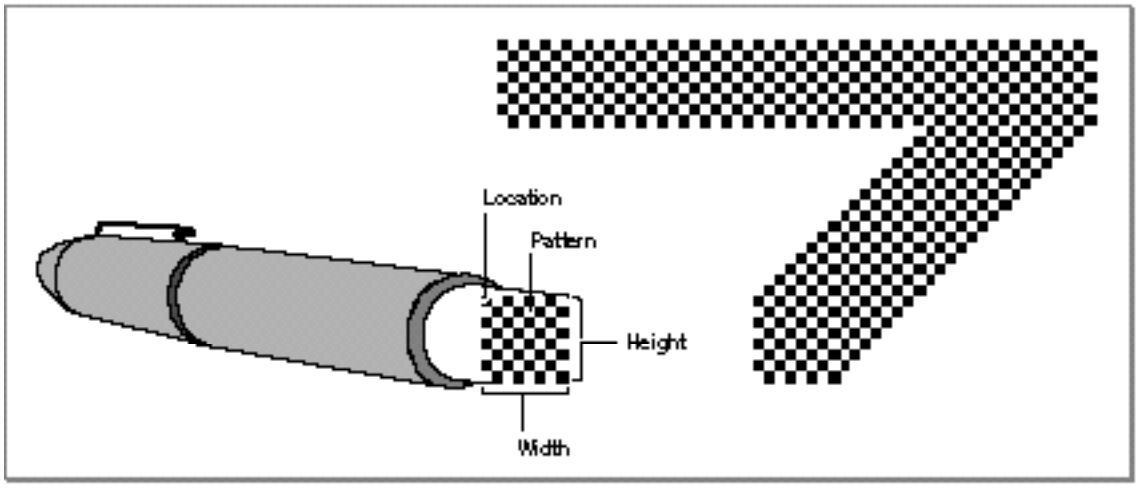
Every graphics port contains one, and only one, graphics pen with which to perform drawing operations. You use this metaphorical pen to draw lines, shapes, and text. Using QuickDraw routines, you can set these five characteristics of the graphics pen for the current graphics port:

- visibility, as stored in the `pnVis` fields of the `GrafPort` and `CGrafPort` records
- size, as stored in the `pnSize` fields of the `GrafPort` and `CGrafPort` records
- location, as stored in the `pnLoc` fields of the `GrafPort` and `CGrafPort` records
- pattern, as stored in the `pnPat` field of the `GrafPort` and `CGrafPort` records
- pattern mode, as stored in the `pnMode` fields of `GrafPort` and `CGrafPort` records

The visibility of the graphics pen simply determines whether the pen draws on the screen. You can use the `HidePen` and `ShowPen` procedures to change the pen’s visibility.

The graphics pen is rectangular in shape, and its size (that is, its height and width) are measured in pixels. The default size is a 1-by-1 pixel square, but you can use the `PenSize` procedure to change its shape from a 0-by-0 pixel square to a 32,767-by-32,767 pixel square. If you set either the width or the height to 0, however, the graphics pen does not draw. (Heights or widths of less than 0 are undefined.) Figure 3-1 illustrates a graphics pen of 8 pixels by 8 pixels.

Figure 3-1 A graphics pen



The graphics pen can be located anywhere on the local coordinate plane of the graphics port, and there are no restrictions on the movement or placement of the pen. You can use the `MoveTo` and `Move` procedures to change the pen's location, which is defined by the point that positions the upper-left corner of the pen. You can use the `GetPen` procedure to determine the pen's current location. As shown in Figure 3-1, the pen draws below and to the right of the point specifying its location.

The pattern and pattern mode determine how the bits under the pen are affected when your application draws lines or shapes. A bit pattern is a repeating 8-by-8 bit image, such as that shown in Figure 3-1. You can use the `PenPat` procedure to change the bit pattern for the graphics pen. Bit patterns are described in more detail in the next section. The pattern mode for the graphics pen determines how the bit pattern interacts with the existing bit image according to one of eight Boolean operations, as described in detail in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8. You can use the `PenMode` procedure to change the pattern mode of the graphics pen.

To determine the size, location, pattern, and pattern mode of the graphics pen, you can use the `GetPenState` procedure, which returns a `PenState` record that contains fields for each of these characteristics. If you need to temporarily change these characteristics, you can use the `SetPenState` procedure to restore the graphics pen to the state saved in the record returned by `GetPenState`.

Upon the creation of a graphics port, QuickDraw assigns these initial values to the graphics pen: a size of (1,1), a pattern of all-black pixels, and a pattern mode of `patCopy`. After changing any of these values, you can use the `PenNormal` procedure to return these initial values to the graphics pen.

"Lines and Shapes" beginning on page 3-11 describes how to use the graphics pen to draw lines and shapes.

Bit Patterns

A **bit pattern** is a 64-pixel image, organized as an 8-by-8 pixel square, that defines a repeating design (such as stripes) or a tone (such as gray). The patterns defined in bit patterns are usually black and white, although any two colors can be used on a color screen. Pixel patterns (which are supported only in color graphics ports) define color patterns at any pixel depth. (Pixel patterns are described in the chapter "Color QuickDraw" in this book.) Figure 3-2 shows a typical bit pattern—the one used for the standard gray desktop pattern on most Macintosh computers with black-and-white screens.

Figure 3-2 A bit pattern

You can use bit patterns to draw lines and shapes on the screen. In a basic graphics port, the graphics pen has a pattern specified in the `pnPat` field of its `GrafPort` record. This bit pattern acts like the ink in the pen; the bits in the pattern interact with the pixels in the bitmap according to the pattern mode of the graphics pen. When you use the `FrameRect`, `FrameRoundRect`, `FrameArc`, `FramePoly`, `FrameRgn`, `PaintRect`, `PaintRoundRect`, `PaintArc`, `PaintPoly`, and `PaintRgn` procedures to draw shapes, these procedures draw the shape with the bit pattern specified in the `pnPat` field.

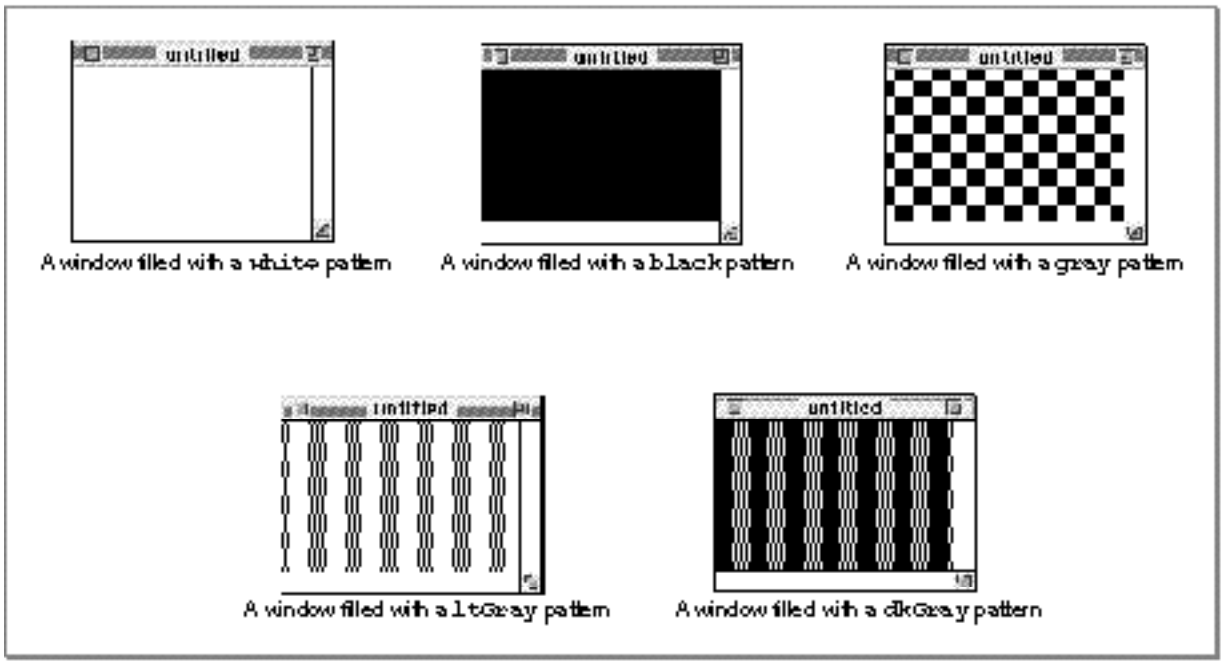
You can use the `FillRect`, `FillRoundRect`, `FillArc`, `FillPoly`, and `FillRgn` procedures to draw shapes with a bit pattern other than that specified in the `pnPat` field of the graphics port. When your application uses one of these procedures, the procedure stores the pattern your application specifies in the `fillPat` field of the `GrafPort` record (or its handle in the `fillPixPat` field of a `CGrafPort` record) and then calls a low-level drawing routine that gets the pattern from that field.

Each graphics port also has a background pattern that's used when an area is erased (such as by using the `EraseRect`, `EraseRoundRect`, `EraseArc`, `ErasePoly`, and `EraseRgn` procedures) and when pixels are scrolled out of an area (such as by using the `ScrollRect` procedure as described in the chapter "Basic QuickDraw"). Every basic graphics port stores a background bit pattern in the `bkPat` field of its `GrafPort` record. (Color graphics ports store a handle to the background pattern in their `bkPixPat` field.)

So that adjacent areas of the same pattern form a continuous, coordinated pattern, all patterns are always drawn relative to the origin of the graphics port.

A basic graphics port supports only bit patterns. Bit patterns are defined in data structures of type `Pattern`, in which each pixel is represented by a single bit. Five such bit patterns are predefined as global variables for your use. These patterns are illustrated in Figure 3-3.

Figure 3-3 Windows filled with the predefined bit patterns



The upper-left window in this figure is filled with the predefined pattern `white`, in which every pixel is white. By default, this is the **background pattern** for a graphics port; that is, this is the pattern displayed when an area is erased or when bits are scrolled out of it. The `white` pattern can also produce useful effects when transferred with an appropriate pattern mode to an existing bit image. (Pattern modes are explained in “Boolean Transfer Modes With 1-Bit Pixels” beginning on page 3-8.)

The middle window in the top row of Figure 3-3 is filled with the predefined bit pattern `black`, in which every pixel is black. This is the initial pattern that QuickDraw assigns to the graphics pen.

Figure 3-3 illustrates a window filled with the predefined pattern `gray`, which uses a combination of black and white pixels. As illustrated in this figure, fewer black pixels in the combination produce the predefined pattern `ltGray`, and more black pixels produce the predefined pattern `dkGray`.

These predefined patterns use colored pixels to produce similar effects in color graphics ports, as described in the chapter "Color QuickDraw."

You can create your own bit patterns in your program code, but it's usually simpler and more convenient to store them in resources of type `'PAT '` or `'PAT#'` and to read them in when you need them. The five predefined patterns are available not only through the global variables provided by QuickDraw but also as system resources stored in the system resource file. You can use the `GetPattern` function and the `GetIndPattern` procedure to get patterns stored as resources.

The result of the transfer of a pattern to a bitmap depends on the pattern mode, which is described next.

Boolean Transfer Modes With 1-Bit Pixels

A **Boolean transfer mode** describes an interaction between the pixels that your application draws and the pixels that are already in the destination bitmap—for example, when you draw a patterned line into a graphics port. Black-and-white drawing uses two types of Boolean transfer modes:

- **source modes** for copying bit images or drawing text
- **pattern modes** for drawing lines and shapes

Color QuickDraw uses Boolean transfer mode differently than basic QuickDraw. Color QuickDraw also has transfer modes that perform arithmetic operations on the red, green, and blue values of color pixels. Using transfer modes with Color QuickDraw is described in the chapter "Color QuickDraw" in this book.

Your application uses source modes when using `CopyBits` procedure (described in "Copying Bits Between Graphics Ports" beginning on page 3-32) and the `CopyDeepMask` procedure (described in the chapter "Color QuickDraw").

Your application uses pattern modes to transfer patterns to lines and shapes. The `penMode` field of a graphics port stores the pattern mode for the graphics pen. You use the pattern mode to draw lines, rectangles, rounded rectangles, ovals, arcs, wedges, polygons, and regions, as follows:

- When you use a procedure like `LineTo`, `FrameRect`, or `FrameOval`, the procedure draws the lines of your shape with the pattern specified in the `pnPat` field of the graphics port, but the procedure transfers that pattern into the graphics port by using the pattern mode specified in the `pnMode` field of the current graphics port.
- When you use a procedure like `PaintRect` or `PaintOval`, the procedure draws your shape with the pattern specified in the `pnPat` field by transferring the pattern with the pattern mode specified in the `pnMode` field.
- When you use a procedure like `FillRect` or `FillOval`, the procedure draws your shape with the pattern you request and uses the `patCopy` pattern mode (which copies your requested pattern directly into the shape).

You use the source mode when using the `CopyBits` procedure to copy a bit image from one graphics port to another and when drawing text using the QuickDraw routines described in the chapter "QuickDraw Text" in *Inside Macintosh: Text*. (The source mode for text is stored in the `textMode` field of a graphics port.)

For both pattern and source modes there are four Boolean operations: COPY, OR, XOR (for exclusive-or), and BIC (for bit clear). Each of these operations has an inverse variant in which the pattern or source is inverted before the transfer, so in fact there are eight operations in all.

The eight operations in the pattern and source modes have names defined as constants. Their effects on 1-bit destination pixels are summarized in Table 3-1. (See the chapter "Color QuickDraw" for information about the effects of these operations on colored pixels—that is, those with a pixel depth of more than 1 pixel.)

Table 3-1 Effect of Boolean transfer modes on 1-bit pixels

Pattern mode	Source mode	Action on destination pixel	
		If pattern or source pixel is black	If pattern or source pixel is white
`patCopy`	`srcCopy`	Force black	Force white
`notPatCopy`	`notSrcCopy`	Force white	Force black
`patOr`	`srcOr`	Force black	Leave alone
`notPatOr`	`notSrcOr`	Leave alone	Force black
`patXor`	`srcXor`	Invert	Leave alone
`notPatXor`	`notSrcXor`	Leave alone	Invert
`patBic`	`srcBic`	Force white	Leave alone
`notPatBic`	`notSrcBic`	Leave alone	Force white

The COPY operations completely replace the pixels in the destination bitmap with either the pixels in the pattern (for the `patCopy` mode) or the pixels in the source bitmap (for the `srcCopy` mode). The inverse COPY operations completely replace the pixels in the destination bitmap with a "photographic negative" of the pattern (for the `notPatCopy` mode) or the source bitmap (for the `notSrcCopy` mode).

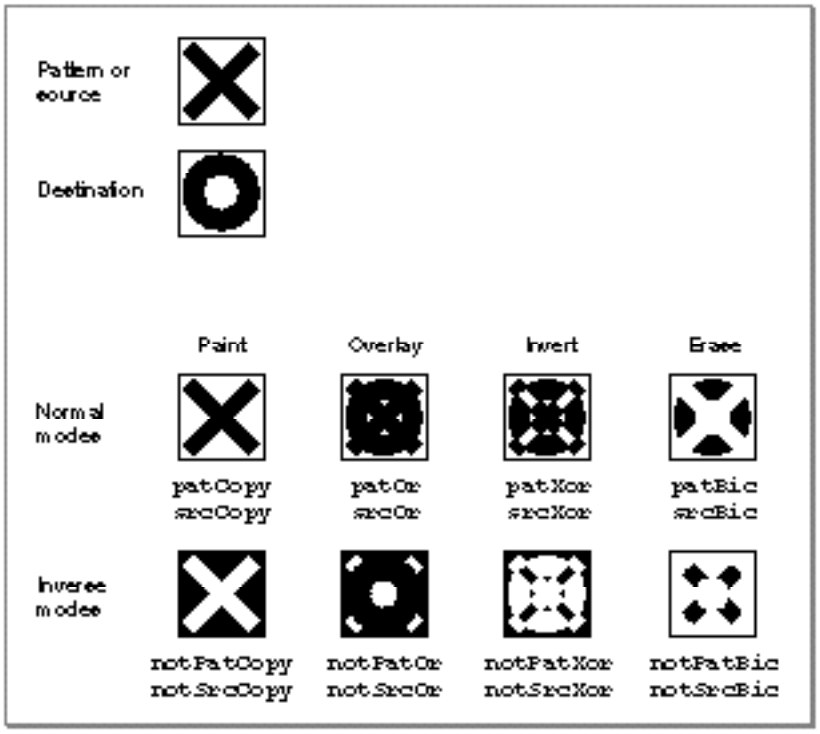
The OR operations add the black pixels from either the pattern (for the `patOr` mode) or the source bitmap (for the `srcOr` mode) to the destination bitmap. The inverse OR operations (`notPatOr` and `notSrcOr` modes) take a "photographic negative" of the pattern or the source bitmap, and then add the black pixels from this negative to the destination bitmap.

The XOR operations (`patXor` and `srcXor` modes) invert the pixels in the destination bitmap that correspond to black pixels in the pattern or source bitmap. The inverse XOR operations (`notPatXor` and `notSrcXor` modes) invert the pixels in the destination bitmap that correspond to white pixels in the pattern or source bitmap.

The BIC operations (`patBic` and `srcBic` modes) turn pixels in the destination bitmap white when they correspond to black pixels in the pattern or source bitmap. The inverse BIC operations (`notPatBic` and `notSrcBic` modes) turn pixels in the destination bitmap white when they correspond to white pixels in the pattern or source bitmap.

These actions are illustrated in Figure 3-4, where a black X is transferred to a destination bitmap consisting of a black O.

Figure 3-4 Examples of Boolean transfer modes



On computers running System 7, you can add dithering to any source mode by adding the following constant or the value it represents to the source mode:

```
CONST ditherCopy  = 64;
```

Dithering mixes existing colors to create the effect of additional colors on indexed devices. It also improves images that you shrink or that you copy from a direct pixel device to an indexed device. Using dithering even when shrinking 1-bit images between basic graphics ports can produce much better representations of the original images. The `CopyBits` procedure always dithers images when shrinking them between pixel maps on direct devices. Dithering is explained in the chapter "Color QuickDraw."

The next section describes how your application uses pattern modes to transfer patterns to lines and shapes.

Lines and Shapes

As explained in the chapter "Basic QuickDraw," rectangles and regions are mathematical models that QuickDraw defines as data types. However, they also can be graphic elements that appear on the screen. A rectangle, for example, can mathematically define a visible area, but it can also be an object to draw.

Defining Lines and Shapes

You use two points to define a line. Using the `LineTo` and `Line` procedures, you can draw lines onscreen using the size, pattern, and pattern mode of the graphics pen for the current graphics port. You can also define a rectangle with two points (the upper-left and lower-right corners of the rectangle) or with four boundary coordinates (one for each side of the rectangle). Using the `FrameRect` procedure, you can draw rectangles that are framed by lines rendered with the size, pattern, and pattern mode of the graphics pen.

You use rectangles to define ovals and rounded rectangles. Rectangles used to define other shapes are called **bounding rectangles.** The lines of bounding rectangles completely enclose the shapes they bound; in other words, no pixels from these shapes lie outside the infinitely thin lines of the bounding rectangles.

Ovals are circular or elliptical shapes defined by the height and width of their bounding rectangles, and rounded rectangles are rectangles with rounded corners defined by the width and height of the ovals forming their corners. Using the `FrameOval` and `FrameRoundRect` procedures, you can draw, respectively, framed ovals and framed rounded rectangles.

You can use rectangles to define ovals that, in turn, you can use to define arcs and wedges. An arc is a portion of an oval's circumference bounded by a pair of radii. A wedge is a pie-shaped segment of an oval. The wedge starts at the center of the oval, is bounded by a pair of radii, and extends to the oval's circumference. You use the `FrameArc` procedure to draw a framed arc, and you use the `PaintArc` or `FillArc` procedure to draw a wedge.

You use lines to define a polygon. First, however, you must call the `OpenPoly` function and then some number of `LineTo` procedures to create lines from the first vertex of the polygon to the second, from the second to the third, and so on, until you've created a line to the last vertex. You then use the `ClosePoly` procedure, which completes the figure by drawing a connecting line from the last vertex back to the first. After defining a polygon in this way, you can draw a framed outline of it using the `FramePoly` procedure.

To define a region, you can use any set of lines or shapes, including other regions, so long as the region's outline consists of one or more closed loops. First, however, you must call the `NewRgn` function and `OpenRgn` procedure. You then use line-, shape-, or region-drawing commands to define the region, which can be concave or convex, can consist of one connected area or many separate ones, and can even have holes in the middle. When you are finished collecting commands to define the outline of the region, you use the `CloseRgn` procedure. You can then draw a framed outline of the region using the `FrameRgn` procedure.

Framing Shapes

Using the `FrameRect`, `FrameOval`, `FrameRoundRect`, `FrameArc`, `FramePoly`, or `FrameRgn` procedure to **frame** a shape draws just its outline, using the size, pattern, and pattern mode of the graphics pen for the current graphics port. The interior of the shape is unaffected, allowing previously existing pixels in the bit image to show through.

Painting and Filling Shapes

Using the `PaintRect`, `PaintOval`, `PaintRoundRect`, `PaintArc`, `PaintPoly`, or `PaintRgn` procedure to **paint** a shape draws both its outline and its interior with the pattern of the graphics pen, using the pattern mode of the graphics pen.

Using the `FillRect`, `FillOval`, `FillRoundRect`, `FillArc`, `FillPoly`, or `FillRgn` procedure to **fill** a shape draws both its outline and its interior with any pattern you specify. The procedure transfers the pattern with the `patCopy` pattern mode, which directly copies your requested pattern into the shape.

Erasing Shapes

Using the `EraseRect`, `EraseOval`, `EraseRoundRect`, `EraseArc`, `ErasePoly`, or `EraseRgn` procedure to **erase** a shape draws both its outline and its interior with the background pattern for the current graphics port. The background pattern is typically solid white on a black-and-white monitor or a solid background color on a color monitor. Making the shape blend into the background pattern of the graphics port effectively erases the shape.

Inverting Shapes

Using the `InvertRect`, `InvertOval`, `InvertRoundRect`, `InvertArc`, `InvertPoly`, or `InvertRgn` procedure to **invert** a shape reverses the colors of all pixels within its boundary. On a black-and-white monitor, this changes all the black pixels in the shape to white and changes all the white pixels to black.

The inversion procedures were designed for 1-bit images in basic graphics ports. These procedures operate on color pixels in color graphics ports, but the results are predictable only with direct devices or 1-bit pixel maps.

For indexed pixels, Color QuickDraw performs the inversion on the pixel indexes. The results depend entirely on the contents of the video device's color lookup table (CLUT). (The CLUT is described in the chapter "Color QuickDraw.")

The eight colors used in basic QuickDraw are stored in a color table represented by the global variable `QDColors`. To display those eight basic QuickDraw colors on an indexed device, Color QuickDraw uses the Color Manager to obtain indexes to the colors in the CLUT that best map to the colors in the `QDColors` color table. Because the index, not the color value, is inverted, the results are again unpredictable. (The eight-color system is described in "The Eight Basic QuickDraw Colors" beginning on page 3-14.)

Inversion works better for direct devices. Inverting a pure green, for example, that has red, green, and blue component values of $0000, $FFFF, and $0000 results in magenta, which has component values of $FFFF, $0000, and $FFFF.

Other Graphic Entities

"Drawing With QuickDraw" beginning on page 3-16 provides an introduction to creating and drawing lines, rectangles, rounded rectangles, ovals, arcs, wedges, polygons, and regions. You can also use QuickDraw routines to draw pictures, cursors, icons, and text.

A QuickDraw picture is the recorded transcription of a sequence of drawing operations that can be played back with the `DrawPicture` procedure. See the chapter "Pictures" for information about creating and displaying QuickDraw pictures.

A cursor is a 16-by-16 pixel image that maps the user's movement of the mouse to relative locations on the screen. An icon is an image (usually 32 by 32 or 16 by 16 pixels) that represents an object, a concept, or a message. For example, the Finder uses icons to represent files and disks. Cursors and icons are stored as resources. See the chapter "Cursor Utilities" for information about drawing cursors. See the chapter "Icon Utilities" in *Inside Macintosh: More Macintosh Toolbox* for information about drawing icons.

See the chapter "QuickDraw Text" in *Inside Macintosh: Text* for information about using QuickDraw routines to draw text.

The Eight Basic QuickDraw Colors

On a color screen, you can draw with colors, even when you are using a basic graphics port. Although basic QuickDraw graphics routines were designed for black-and-white drawing, they also support the **eight-color system** that basic QuickDraw predefines for display on color screens and color printers. Because Color QuickDraw also supports this system, it is compatible across all Macintosh platforms. (This section describes the rudimentary color capabilities included in basic QuickDraw. See the next chapter, "Color QuickDraw," for information about more sophisticated color use in your application.)

A pair of fields in a graphics port, `fgColor` and `bkColor`, specify a foreground and background color. The **foreground color** is the color used for bit patterns and for the graphics pen when drawing. By default, the foreground color is black. The **background color** is the color of the pixels in the bitmap wherever no drawing has taken place. By default, the background color is white. However, you can use the `ForeColor` and `BackColor` procedures to change these fields. (When printing, however, use the `ColorBit` procedure to set the foreground color.) For example, on a color screen the following lines of code draw a red rectangle against a blue background.

```
BackColor(blueColor);   {make a blue background}
ForeColor(redColor);    {draw with red ink}
PenMode(patCopy);       {when drawing, replace background color }
                        { with ink's color}
PaintRect(20,20,80,80); {create and paint the red rectangle}
```

If you use the `OpenCPicture` or `OpenPicture` function to include this code in a picture definition, these colors are stored in the picture. However, basic QuickDraw cannot store these colors in a bitmap. See the chapter "Pictures" in this book for more information about defining and drawing pictures.

The basic QuickDraw color values consist of 1 bit for normal black-and-white drawing (black on white), 1 bit for inverted black-and-white drawing (white on black), 3 bits for the additive primary colors (red, green, blue) used in video display, and 4 bits for the subtractive primary colors (cyan, magenta, yellow, black) used in printing. QuickDraw includes a set of predefined constants for those standard colors:

```
CONST
  whiteColor   = 30;
  blackColor   = 33
  yellowColor  = 69;
  magentaColor = 137;
  redColor     = 205;
  cyanColor    = 273;
  greenColor   = 341;
  blueColor    = 409;
```

These are the only colors available in basic QuickDraw (or with Color QuickDraw drawing into a basic graphics port). When you specify these colors on a Macintosh computer with Color QuickDraw, Color QuickDraw draws these colors if the user has set the screen to a color mode.

These eight color values are based on a planar model: each bit position corresponds to a different color plane, and the value of each bit indicates whether a particular color plane should be activated. (The term *color plane* refers to a logical plane, rather than a physical plane.) The individual color planes combine to produce the full-color image.

There are three advantages to using basic QuickDraw's color system:

- It is available across all platforms, so you don't have to check for the presence of Color QuickDraw.
- It is much simpler to use than Color QuickDraw.
- It works well on an ImageWriter printer with a color ribbon.

The main disadvantage is that basic QuickDraw is limited to eight predefined colors. Another problem is that, if the graphics port in which you are working happens to be a color graphics port, then the two color systems may clash. For example, saving the current foreground color (from the `fgColor` field of the color graphics port) and then later restoring it with the `ForeColor` procedure doesn't work: the original content of the `fgColor` field is an index value for a color graphics port using indexed colors. This index value is not what basic QuickDraw's `ForeColor` procedure expects as a parameter.

In System 7, these Color QuickDraw routines are available to basic QuickDraw: `RGBForeColor`, `RGBBackColor`, `GetForeColor`, and `GetBackColor`. Described in the next chapter, "Color QuickDraw," these routines can also assist you in manipulating the eight-color system of basic QuickDraw. When running on a System 7 computer, your application should use `GetForeColor` and `GetBackColor` to determine the foreground color and background color instead of checking the `fgColor` and `bkColor` fields of the `GrafPort` record.

The next section provides an introduction to creating and drawing lines and shapes. Without using a color graphics port, you can use the `ForeColor` or `RGBForeColor` procedure on a color screen to draw these lines and shapes in color, against the background color you set with the `BackColor` or `RGBBackColor` procedure.

Drawing With QuickDraw

You can use QuickDraw's basic drawing routines to

- draw lines of various thicknesses and in various patterns
- draw rectangles, rounded rectangles, ovals, arcs, wedges, polygons, and regions in various patterns
- draw lines and shapes in any of eight predefined colors, against a background of any of these eight predefined colors
- perform calculations on and manipulate rectangles and regions
- copy bits from the bit image in the bitmap of one graphics port into the bitmap of another graphics port
- customize QuickDraw's drawing behavior

System software uses QuickDraw's drawing routines to implement the Macintosh user interface. The next several sections provide an introduction to these routines, which your application can use to create complex onscreen images.

To draw lines, your application

- moves the graphics pen to a location within its graphics port
- draws a line to a different coordinate

To draw rectangles, rounded rectangles, ovals, arcs, and wedges, your application generally

- defines the outline of the shape in the local coordinates of the graphics port
- frames the shape's outline to draw it
- transfers patterns to the outline and interior of the shape to paint or fill it

To draw regions and polygons, your application

- uses an open routine to start building the shape
- calls drawing routines to build the shape
- uses a close routine to stop collecting drawing routines for the shape
- frames the shape's outline to draw it
- transfers patterns to the outline and interior of the shape to paint or fill it

These tasks are explained in greater detail in the rest of this chapter.

Before using QuickDraw's drawing routines, you must initialize QuickDraw with the `InitGraf` procedure, as explained in the chapter "Basic QuickDraw."

The routines described in this chapter are available on all models of Macintosh computers. However, all nonwhite colors that you specify with the `ForeColor` and `BackColor` procedures are displayed as black on a black-and-white screen. Before using the `ForeColor` and `BackColor` procedures to display colors in a basic graphics port, you can use the `DeviceLoop` procedure, which is described in the chapter "Graphics Device," to determine the color characteristics of the current screen.

Drawing Lines

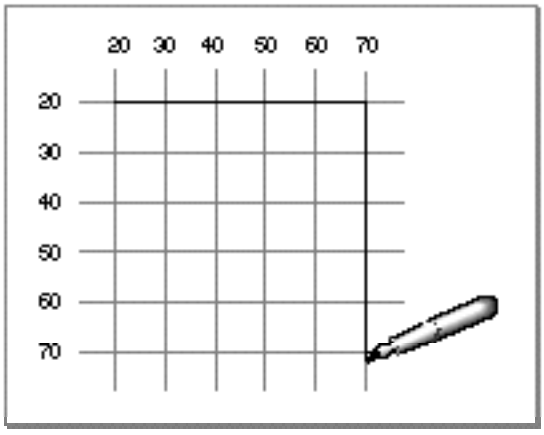
A line is defined by two points: the current location of the graphics pen and its destination. The graphics pen draws below and to the right of the defining points. As described in "The Graphics Pen" on page 3-4, the pen draws the line between its defining points with the size, pattern, and pattern mode stored in the current graphics port.

You specify where to begin drawing a line by using the `MoveTo` or `Move` procedure to place the graphics pen at some point in the window's local coordinate system. Then you call the `LineTo` or `Line` procedure to draw a line from there to another point. Take, for example, the following lines of code:

```
MoveTo(20,20);
LineTo(70,20);
LineTo(70,70);
```

The `MoveTo` procedure moves the graphics pen to a point with a horizontal coordinate of 20 and a vertical coordinate of 20 (in the local coordinate system of the graphics port). The first call to the `LineTo` procedure draws a line from that position to the point with a horizontal coordinate of 70 and a vertical coordinate of 20. The second call to the `LineTo` procedure draws a line from the pen's new position to the point with a horizontal coordinate of 70 and a vertical coordinate of 70, as shown in Figure 3-5.

Figure 3-5 Using the `LineTo` procedure

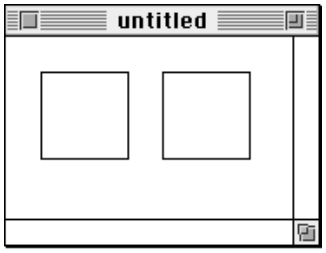


Listing 3-1 illustrates how to use the `LineTo` procedure to draw the four sides of a square, which is shown on the left side of Figure 3-6. In Figure 3-6, the current graphics port is the window "untitled."

Listing 3-1 Drawing lines with the `LineTo` and `Line` procedures

```
PROCEDURE MyDrawLines;
BEGIN
   MoveTo(20,20);
   LineTo(70,20);
   LineTo(70,70);
   LineTo(20,70);
   LineTo(20,20);

   Move(70,0);
   Line(50,0);
   Line(0,50);
   Line(-50,0);
   Line(0,-50);
END;
```

Figure 3-6 Drawing lines

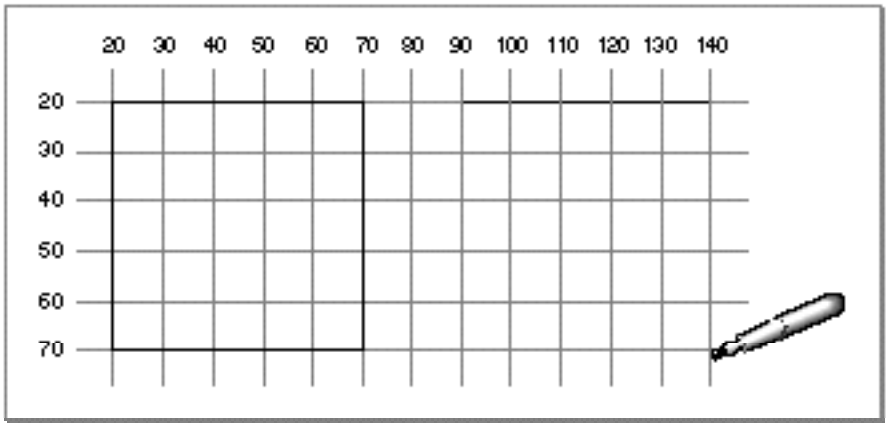


The `MoveTo` and `LineTo` procedures require you to specify a point in the local coordinate system of the current graphics port. These procedures then transfer the graphics pen to that specific location. As alternatives to using the `MoveTo` and `LineTo` procedures, you can use the `Move` and `Line` procedures, which require you to pass relative horizontal and vertical distances to move the pen from its current location. The square on the right side of Figure 3-6 is drawn using the `Move` and `Line` procedures.

The final call to `LineTo` in Listing 3-1 moves the graphics pen to the point with a horizontal coordinate of 20 and a vertical coordinate of 20. Listing 3-1 then uses the `Move` procedure to move the graphics pen a horizontal distance of 70 points—that is, to the point with a horizontal coordinate of 90. The first call to the `Line` procedure draws a

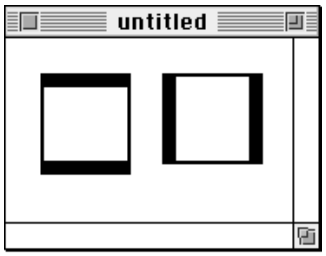
horizontal line 50 pixels long—that is, to the point with a horizontal coordinate of 140 and a vertical coordinate of 20. Starting from there, the second call to `Line` draws a vertical line 50 pixels long—that is, to the point with a horizontal coordinate of 140 and a vertical coordinate of 70, as shown in Figure 3-7.

Figure 3-7 Using the `LineTo` and `Line` procedures



In Figure 3-6, the lines are drawn using the default pen size (1,1), giving the line a vertical depth of one pixel and a horizontal width of one pixel. You can use the `PenSize` procedure to change the width and height of the graphics pen so that it draws thicker lines, as shown in Figure 3-8.

Figure 3-8 Resizing the pen



The square on the left side of Figure 3-8 is drawn with a pen that has a width of two pixels and a height of eight pixels. The square on the right side of this figure is drawn with a pen that has a width of eight pixels and a height of two pixels. Listing 3-2 shows the code that draws these squares.

Listing 3-2 Using the PenSize procedure

```
PROCEDURE MyResizePens;
BEGIN
   PenSize(2,8);
   MoveTo(20,20);
   LineTo(70,20); LineTo(70,70); LineTo(20,70); LineTo(20,20);
   PenSize(8,2);
   Move(70,0);
   Line(50,0); Line(0,50); Line(-50,0); Line(0,-50);
   PenNormal;
END;
```

At the end of Listing 3-2, the PenNormal procedure is used to restore the graphics pen to its default size, pattern, and pattern mode.

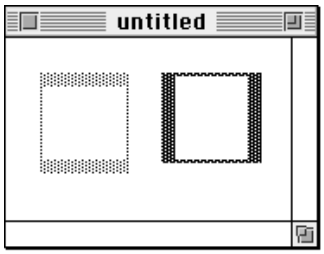
The default pattern for the graphics pen consists of all black pixels. However, you can use the PenPat procedure to change the pen's pattern. When you use the PenPat procedure, you can pass it any one of the predefined global variables listed in Table 3-2 to specify the bit pattern for the graphics pen.

Table 3-2 The global variables for five predefined bit patterns

Global variable	Result
black	All-black pattern
dkGray	75% gray pattern
gray	50% gray pattern
ltGray	25% gray pattern
white	All-white pattern

In Figure 3-9, the pen pattern for the square on the left has changed to `ltGray`; the pen pattern for the square on the right has changed to `dkGray`.

Figure 3-9 Changing the pen pattern



Listing 3-3 shows the code that produces these squares.

Listing 3-3 Using the `PenPat` procedure to change the pattern of the graphics pen

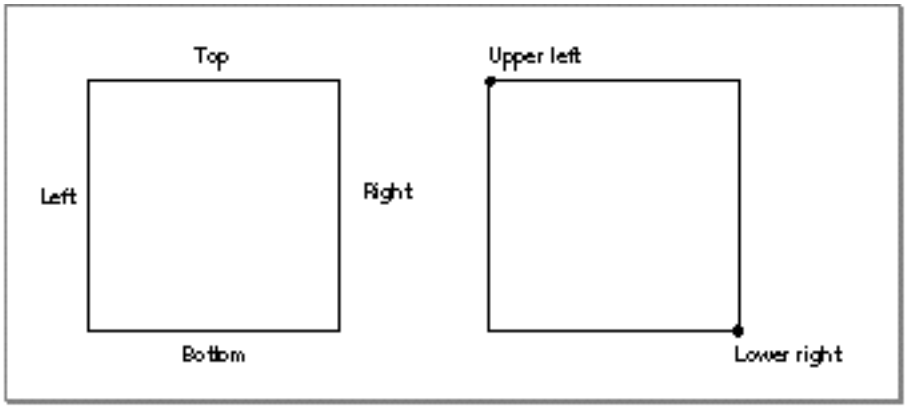
```
PROCEDURE MyRepatternPens;
BEGIN
   PenSize(2,8);
   PenPat(ltGray);
   MoveTo(20,20);
   LineTo(70,20); LineTo(70,70); LineTo(20,70); LineTo(20,20);

   PenSize(8,2);
   PenPat(dkGray);
   Move(70,0);
   Line(50,0); Line(0,50); Line(-50,0); Line(0,-50);
   PenNormal;
END;
```

QuickDraw provides methods for drawing squares and rectangles that are easier than drawing each side individually as a line. The next section describes how to draw rectangles.

Drawing Rectangles

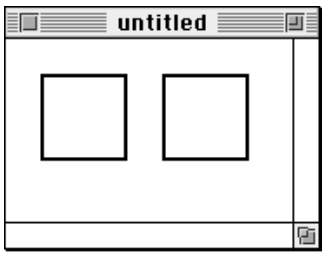
As explained in the chapter "Basic QuickDraw," rectangles are mathematical entities. There are two ways to specify a rectangle: by its four boundary coordinates, as shown in the left rectangle in Figure 3-10, or by its upper-left and lower-right points, as shown in the right rectangle.

Figure 3-10 Two ways to specify a rectangle



However, specifying a rectangle does not draw one. Because the border of a rectangle is infinitely thin, it can have no direct representation on the screen until you use the `FrameRect` procedure to draw its outline, or you can use the `PaintRect` or `FillRect` procedure to draw its outline and its interior with a pattern. Figure 3-11 illustrates two rectangles that are drawn with the `FrameRect` procedure.

Figure 3-11 Drawing rectangles



Listing 3-4 shows the code that draws the rectangles in Figure 3-11. This listing uses the `PenSize` procedure to assign a size of (2,2) to the graphics pen. Then the code assigns four boundary coordinates to the rectangle on the left side of this figure, and it calls `FrameRect` to use the graphics pen to draw the rectangle's outline.

Listing 3-4 Using the `FrameRect` procedure to draw rectangles

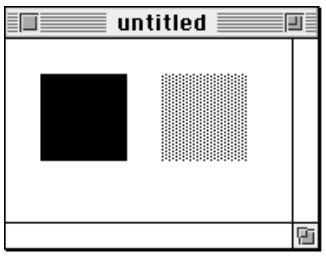
```
PROCEDURE MyDrawRects;
   VAR
      firstRect, secondRect: Rect;
BEGIN
   PenSize(2,2);

   firstRect.top := 20;
   firstRect.left := 20;
   firstRect.bottom := 70;
   firstRect.right := 70;
   FrameRect(firstRect);

   SetRect(secondRect,90,20,140,70);
   FrameRect(secondRect);

   PenNormal;
END;
```

To shorten code text, Listing 3-4 uses the `SetRect` procedure to define the rectangle on the right side of Figure 3-11. Again, `FrameRect` is used to draw an outline around the rectangle. Notice that while a `Rect` record lists the fields for a rectangle's boundaries in the order `top`, `left`, `bottom`, and `right`, you pass these boundaries as parameters to the `SetRect` procedure in the order `left`, `top`, `right`, and `bottom`.

Remember that the graphics pen hangs to the right of and below its location point; therefore, the lower-right corner of the two-pixel outline around the rectangle on the right side of Figure 3-11 lies at the point with a horizontal coordinate of 142 and a vertical coordinate of 72.

Figure 3-12 illustrates painted and filled rectangles. Listing 3-5 shows the code that creates these images.

Figure 3-12 Painting and filling rectangles



Listing 3-5 uses the `PaintRect` procedure to draw the outline and the interior of the rectangle on the left side of Figure 3-12 with the pattern of the graphics pen, according to the pattern mode of the graphics pen. Because Listing 3-5 calls the `PenNormal` procedure immediately before calling `PaintRect`, the graphics pen has its default characteristics: a pattern of all-black pixels and the `patCopy` pattern mode, which changes all of the pixels in the destination to the pen's pattern.

Listing 3-5 Using the `PaintRect` and `FillRect` procedures

```
PROCEDURE MyPaintAndFillRects;
   VAR
   firstRect, secondRect: Rect;
BEGIN
   PenNormal;
   SetRect(firstRect,20,20,70,70);
   PaintRect(firstRect);
   SetRect(secondRect,20,90,70,140);
   FillRect(secondRect,ltGray);
END;
```

The `PaintRect` procedure always uses the pattern and pattern mode of the graphics pen when drawing a rectangle. If you want to use a pattern other than that of the graphics pen, you can use the `FillRect` procedure. The `FillRect` procedure, however, *always* uses the `patCopy` pattern mode. Listing 3-5 uses the `FillRect` procedure to draw the outline and the interior of the rectangle on the right side of Figure 3-12 with a light gray pattern.

Note

Neither the `PaintRect` nor `FillRect` procedure changes the location of the graphics pen. ◆

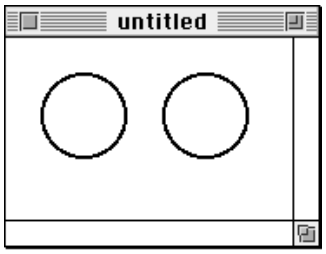
If the application that draws the rectangles in Figure 3-12 uses the `EraseRect` procedure to erase them both, then they would be filled with the background pattern specified by the `bkPat` field of the current graphics port. If the application uses the `InvertRect` procedure to invert the rectangles, then the black pixels in each would become white and the white pixels would become black.

QuickDraw provides a similar set of routines for drawing rounded rectangles, which are defined by their rectangles and the widths and heights of the ovals forming their corners. See "Drawing Rounded Rectangles" beginning on page 3-63 for detailed information about these routines.

Drawing Ovals, Arcs, and Wedges

An oval is a circular or elliptical shape defined by the bounding rectangle that encloses it. After specifying the bounding rectangle for an oval, you use the `FrameOval` procedure to draw its outline, or the `PaintOval` or `FillOval` procedure to draw its outline and its interior with a pattern. Figure 3-13 illustrates two ovals drawn with the `FrameOval` procedure.

Figure 3-13 Drawing ovals



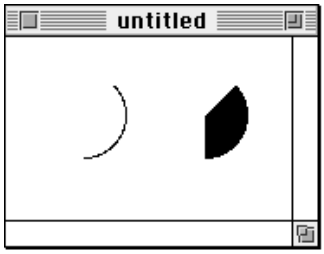
Listing 3-6 shows the code that produces the ovals in Figure 3-13. The bounding rectangles for the ovals are created with the `SetRect` procedure. The resulting rectangles are then passed to the `FrameOval` procedure.

Listing 3-6 Using the `FrameOval` procedure to draw ovals

```
PROCEDURE MyDrawOvals;
   VAR
      firstRect, secondRect: Rect;
BEGIN
   PenSize(2,2);
   SetRect(firstRect,20,20,70,70);   {create a bounding rectangle}
   FrameOval(firstRect);             {draw the oval}
   SetRect(secondRect,90,20,140,70); {create a bounding rectangle}
   FrameOval(secondRect);            {draw the oval}
   PenNormal;
END;
```

An arc is defined as a portion of an oval's circumference bounded by a pair of radii. A wedge is a pie-shaped segment bounded by a pair of radii, and it extends from the center of the oval to its circumference. You use the `FrameArc` procedure to draw an arc (as shown on the left side of Figure 3-14), and you use the `PaintArc` or `FillArc` procedure to draw a wedge (as shown on the right side of Figure 3-14).

Figure 3-14 Drawing an arc and a wedge



Listing 3-7 shows the code that produces the images in Figure 3-14. The `FrameArc`, `PaintArc`, and `FillArc` procedures take three parameters: a rectangle that defines an oval's boundaries, an angle indicating the start of the arc, and an angle indicating the arc's extent. For the angle parameters, 0° indicates a vertical line straight up from the center of the oval. Positive values indicate angles in the clockwise direction from this vertical line, and negative values indicate angles in the counterclockwise direction. The arc and the wedge in Figure 3-14 both begin at 45° and extend to 135°.

Listing 3-7 Using the `FrameArc` and `PaintArc` procedures

```
PROCEDURE MyDrawArcAndPaintWedge;
   VAR
      firstRect, secondRect: Rect;
BEGIN
   SetRect(firstRect,20,20,70,70);   {create a bounding rectangle}
   FrameArc(firstRect,45,135);       {draw an arc}
   SetRect(secondRect,90,20,140,70); {create a bounding rectangle}
   PaintArc(secondRect,45,135);      {draw a wedge}
END;
```

You can also fill, erase, and invert wedges by using, respectively, the `FillArc`, `EraseArc`, and `InvertArc` procedures.

Drawing Regions and Polygons

Before drawing regions and polygons, you must call several routines to create them. To create a region or polygon, you first call an open routine, which tells QuickDraw to collect subsequent routines to construct the shape. You use a close procedure when you are finished constructing the region or polygon. You can then frame the shape, fill it, paint it, erase it, and invert it.

To begin defining a region, you must use the `NewRgn` function to allocate space for it, and then call the `OpenRgn` procedure. You can then use any QuickDraw routine to construct the outline of the region. The outline can be any set of lines and shapes (including other regions) forming one or more closed loops. When you are finished constructing the region, use the `CloseRgn` procedure.

▲ **WARNING**

Ensure that the memory for a region is valid before calling routines to manipulate that region; if there isn't sufficient memory, the system may crash. Regions are limited to 32 KB in size in basic QuickDraw and 64 KB in Color QuickDraw. Before defining a region, you can use the Memory Manager function `MaxMem` to determine whether the memory for the region is valid. You can determine the current size of an existing region by calling the Memory Manager function `GetHandleSize`. (Both `MaxMem` and `GetHandleSize` are described in *Inside Macintosh: Memory*.) When you record drawing operations in an open region, the resulting region description may overflow the 32 KB or 64 KB limit. Should this happen in Color QuickDraw, the `QDError` function (described in the chapter "Color QuickDraw" in this book) returns the result code `regionTooBigError`. ▲

To draw the region, use the `FrameRgn`, `PaintRgn`, or `FillRgn` procedure. To draw the region with the background pattern of the graphics port, use the `EraseRgn` procedure; to invert the pixels in the region, use the `InvertRgn` procedure. When you no longer need the region, use the `DisposeRgn` procedure to release the memory used by the region.

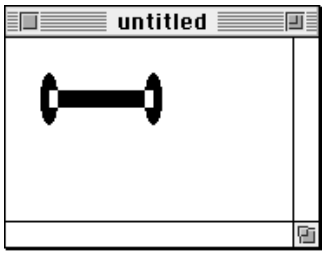
Listing 3-8 illustrates how to create and open a region, define a shape, close the region, fill it with the all-black pattern, and dispose of the region.

Listing 3-8 Creating and drawing a region

```
PROCEDURE MyDrawDumbbell;
VAR
   grow:       LongInt;
   dumbbell:   RgnHandle;
   tempRect:   Rect;
BEGIN
   IF MaxMem(grow) > kMinReserve THEN
   BEGIN
      dumbbell := NewRgn;           {create a new region}
      OpenRgn;                      {begin drawing instructions}
      SetRect(tempRect,20,20,30,50);
      FrameOval(tempRect);          {form the left "weight"}
      SetRect(tempRect,25,30,85,40);
      FrameRect(tempRect);          {form the bar}
      SetRect(tempRect,80,20,90,50);
      FrameOval(tempRect);          {form the right "weight"}
      CloseRgn(dumbbell);           {stop collecting}
      FillRgn(dumbbell,black);      {draw the shape onscreen}
      IF QDError <> noErr THEN
         ; {likely error is that there is insufficient memory}
      DisposeRgn(dumbbell)             {dispose of the region}
   END;
END;
```

Figure 3-15 shows the shape created by Listing 3-8.

Figure 3-15 A shape created by a region



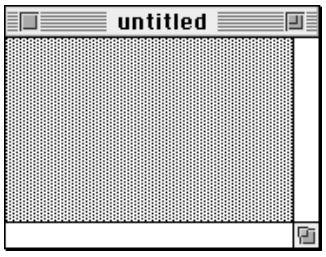
To assist you with scrolling, you can use QuickDraw routines to define a clipping region that excludes the scroll bars of the content region of a window. You can then scroll that area so that the region being updated does not draw into the scroll bars. Listing 3-9 illustrates how to create such a clipping region and, for illustrative purposes, how to fill it with a pattern. (The chapter "Basic QuickDraw" illustrates how to scroll the pixels in a rectangle such as the one created with the `ClipRect` procedure in Listing 3-9.)

Listing 3-9 Creating a clipping region and filling it with a pattern

```
FUNCTION MyFillClipRegion: RgnHandle;
VAR
   grow:          LongInt;
   newClip:       Rect;
   oldClipRegion: RgnHandle;
   newClipRegion: RgnHandle;
   myWindow:      WindowPtr;
BEGIN
   IF MaxMem(grow) > kMinReserve THEN
   BEGIN
      oldClipRegion := NewRgn;   {allocate old clipping region}
      myWindow := FrontWindow;   {get the front window}
      SetPort(myWindow);         {make the front window the current }
                                 { graphics port}
      GetClip(oldClipRegion);    {save the old clipping region}
      newClip := myWindow^.portRect;   {create a new rectangle}
      newClip.right := newClip.right - 15;   {exclude scroll bar}
      newClip.bottom := newClip.bottom - 15; {exclude scroll bar}
      ClipRect(newClip); {make the new rectangle the clipping region}
      newClipRegion := NewRgn;   {allocate new clipping region}
      RectRgn(newClipRegion, newClip);
      FillRgn(newClipRegion, ltGray);  {paint clipping region gray}
      SetClip(oldClipRegion);    {restore previous clipping region}
      IF QDError <> noErr THEN
         ; {likely error is that there is insufficient memory}
      DisposeRgn(oldClipRegion); {dispose previous clipping region}
      MyFillClipRect := (newClipRegion);
   END;
END;
```

Figure 3-16 shows the results of using the code in Listing 3-9.

Figure 3-16 Filling a clipping region



To create a polygon, you first call the `OpenPoly` function and then some number of `LineTo` procedures to draw lines from the first vertex of the polygon to the second, from the second to the third, and so on, until you've drawn a line to the last vertex. You then use the `ClosePoly` procedure, which completes the figure by drawing a connecting line from the last vertex back to the first. After defining a polygon in this way, you can display it with the `FramePoly`, `PaintPoly`, `FillPoly`, `ErasePoly`, and `InvertPoly` procedures. When you are finished using the polygon, use the `KillPoly` procedure to release its memory.

▲ **WARNING**
Do not create a height or width for the polygon greater than 32,767 pixels, or `PaintPoly` will crash. ▲

Listing 3-10 illustrates how to create a triangular polygon and fill it with a gray pattern.

Listing 3-10 Creating a triangular polygon

```
PROCEDURE MyDrawTriangle;
VAR
   triPoly: PolyHandle;
BEGIN
   triPoly := OpenPoly; {save handle and begin collecting lines}
   MoveTo(300,100);     {move to first point}
   LineTo(400,200);     {form the triangle's sides}
   LineTo(200,200);
   ClosePoly;           {stop collecting lines}
   FillPoly(triPoly,gray); {fill the polygon with gray}
   IF QDError <> noErr THEN
      ; {likely error is that there is insufficient memory}
   KillPoly(triPoly);   {dispose of its memory}
END;
```

Performing Calculations and Other Manipulations of Shapes

QuickDraw provides a multitude of routines for manipulating rectangles and regions. You can use the routines that manipulate rectangles to manipulate any shape based on a rectangle—namely, rounded rectangles, ovals, arcs, and wedges. For example, you could define a rectangle to bound an oval and then frame the oval. You could then use the `OffsetRect` procedure to move the oval's bounding rectangle downward. Using the offset bounding rectangle, you could frame a second, connected oval to form a figure eight with the first oval. You could use that shape to help define a region. You could create a second region, and then use the `UnionRgn` procedure to create a region from the union of the two.

The routines for performing calculations and other manipulations of rectangles are summarized in Table 3-3 and are described in detail in "Creating and Managing Rectangles" beginning on page 3-52.

Table 3-3 QuickDraw routines for calculating and manipulating rectangles

Routine	Description
`EmptyRect`	Determines whether a rectangle is an empty rectangle
`EqualRect`	Determines whether two rectangles are equal
`InsetRect`	Shrinks or expands a rectangle
`OffsetRect`	Moves a rectangle
`PtInRect`	Determines whether a pixel is enclosed in a rectangle
`PtToAngle`	Calculates the angle from the middle of a rectangle to a point
`Pt2Rect`	Determines the smallest rectangle that encloses two points
`SectRect`	Determines whether two rectangles intersect
`UnionRect`	Calculates the smallest rectangle that encloses two rectangles

The routines for performing calculations and other manipulations of regions are summarized in Table 3-4 and are described in detail in “Creating and Managing Regions” beginning on page 3-85.

Table 3-4 QuickDraw routines for calculating and manipulating regions

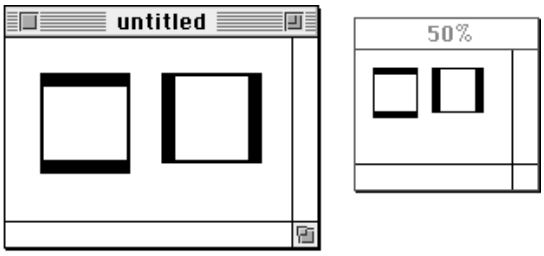
Routine	Description
`CopyRgn`	Makes a copy of a region
`DiffRgn`	Subtracts one region from another
`EmptyRgn`	Determines whether a region is empty
`EqualRgn`	Determines whether two regions have identical sizes, shapes, and locations
`InsetRgn`	Shrinks or expands a region
`OffsetRgn`	Moves a region
`PtInRgn`	Determines whether a pixel is within a region
`RectInRgn`	Determines whether a rectangle intersects a region
`RectRgn`	Changes a region to a rectangle
`SectRgn`	Calculates the intersection of two regions
`SetEmptyRgn`	Sets a region to empty
`SetRectRgn`	Changes a region to a rectangle
`UnionRgn`	Calculates the union of two regions
`XorRgn`	Calculates the difference between the union and the intersection of two regions

Note that while you can use the `OffSetPoly` procedure to move a polygon, QuickDraw provides no other routines for calculating or manipulating polygons.

Copying Bits Between Graphics Ports

You can use the `CopyBits` procedure to copy a bit image from one graphics port to another. Along with the `CopyMask` procedure and the Color QuickDraw procedure `CopyDeepMask`, `CopyBits` is integral to QuickDraw’s image-processing capabilities. You can use `CopyBits` to move offscreen graphics images into an onscreen window, to blend colors for the image in a pixel map, and to shrink and expand images. For example, Figure 3-17 illustrates how `CopyBits` can be used to scale the image in one window to a smaller image in another window.

Figure 3-17 Shrinking images between graphics ports



Listing 3-11 shows the code that produces the scaled image in Figure 3-17.

Listing 3-11 Using the `CopyBits` procedure to copy between two windows

```
PROCEDURE MyShrinkImages;
   VAR
      myWindow:               WindowPtr;
      sourceRect, destRect:   rect;
      halfHeight, halfWidth:  Integer;
BEGIN
   myWindow := FrontWindow;
   sourceRect.top := myWindow^.portRect.top; {create source rectangle}
   sourceRect.left := myWindow^.portRect.left;
   sourceRect.bottom := myWindow^.portRect.bottom - 15; {exclude scroll bar}
   sourceRect.right := myWindow^.portRect.right - 15;  {exclude scroll bar}

   destRect.top := gShrinkWindow^.portRect.top; {create destination rect}
   destRect.left := gShrinkWindow^.portRect.left;
   halfHeight :=              {make destination half as tall as the source}
            Integer((sourceRect.bottom - sourceRect.top)) DIV 2;
   destRect.bottom := destRect.top + halfHeight;
   halfWidth :=               {make destination half as wide as the source}
            Integer((sourceRect.right - sourceRect.left)) DIV 2;
   destRect.right := destRect.left + halfWidth;

   GetPort(myWindow);      {save the graphics port for the active window}
   SetPort(gShrinkWindow); {make the target window the current }
                           { graphics port for drawing purposes}
```

```
   CopyBits(myWindow^.portBits,
            gShrinkWindow^.portBits,
            sourceRect,
            destRect,
            srcCopy+ditherCopy, NIL);
   IF QDError <> noErr THEN
      ; {likely error is that there is insufficient memory}
   SetPort(myWindow);      {restore active window as current graphics port}
END;
```

When copying between basic graphics ports, you specify a source bitmap and a destination bitmap to `CopyBits`. Remember that the bitmap is stored in the `portBits` field of a `GrafPort` record. By dereferencing the desired window record when it calls `CopyBits`, Listing 3-11 uses the bitmap for the front window, "untitled" in Figure 3-17, as the source bitmap. Listing 3-11 uses the bitmap for the window titled "50%" as the destination bitmap.

When copying images between color graphics ports, as explained in the chapter "Color QuickDraw," you must coerce each `CGrafPort` record to a `GrafPort` record, dereference the `portBits` fields of each, and then pass these "bitmaps" to `CopyBits`.

Note
If there is insufficient memory to complete a `CopyBits` operation in Color QuickDraw, the `QDError` function (described in the chapter "Color QuickDraw" in this book) returns the result code –143. ◆

You can specify differently sized source and destination rectangles, and `CopyBits` scales the source image to fit the destination. Listing 3-11 uses the area of the port rectangle excluding the scroll bars as the source rectangle. To scale the image in the front window, Listing 3-11 creates a destination rectangle that is half as high and half as wide as the source rectangle.

The manner by which `CopyBits` transfers the bits between bitmaps depends on the source mode that you specify. In Listing 3-11, the `srcCopy` mode is used to copy bits from the source directly into the destination. Source modes are described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8.

Note
To scale shapes and regions within the same graphics port, you can use the routines described in "Scaling and Mapping Points, Rectangles, Polygons, and Regions" beginning on page 3-104. ◆

To gracefully display complex images that your application creates, your application should use the drawing routines described in this chapter to construct such images in offscreen graphics worlds. Your application can then use the `CopyBits` procedure to transfer these images to onscreen graphics ports. This technique prevents the choppiness that can occur when you build complex images onscreen, and is described in the chapter "Offscreen Graphics Worlds," which also offers an example of using a mask to copy color pixels from an offscreen graphics world.

To copy only certain bits from a bitmap, you can use the `CopyMask` procedure, which is a specialized variant of `CopyBits`. The `CopyMask` procedure, which is described on page 3-119, transfers bits only where the corresponding bits of another bit image, which serves as a mask, are set to 1 (that is, black). The `CopyMask` procedure does not allow scaling or resizing. However, the `CopyDeepMask` procedure, which is described on page 3-120, does allow scaling and resizing; in effect it combines the capabilities of the `CopyBits` and `CopyMask` procedures.

Customizing QuickDraw's Low-Level Routines

For each shape that QuickDraw knows how to draw, there are procedures that perform these basic graphics operations on the shape: frame, paint, erase, invert, and fill. Those procedures in turn call a low-level drawing routine for the shape. For example, the `FrameOval`, `PaintOval`, `EraseOval`, `InvertOval`, and `FillOval` procedures all call the low-level procedure `StdOval`, which draws the oval. For each type of object QuickDraw can draw, including text and lines, there's a pointer to such a low-level routine. By changing these pointers, you can install your own routines, and either completely override the standard ones or call them after your routines have modified their parameters as necessary.

Other low-level routines that you can install in this way include

- The procedure (called by `CopyBits`) that performs bit and pixel transfer.
- The function that measures the width of text and is called by the QuickDraw text routines `CharWidth`, `StringWidth`, and `TextWidth`. (These QuickDraw text routines are described in the chapter "QuickDraw Text" in *Inside Macintosh: Text*.)
- The procedure that processes picture comments. The standard procedure ignores picture comments. (Picture comments are described in Appendix B of this book.)
- The procedure that saves drawing commands as the definition of a picture, and the procedure that retrieves them. These enable your application to draw on remote devices, print to the disk, get picture input from the disk, and support large pictures.

All of the low-level QuickDraw routines that your application can replace or call after performing its own operations are described in "Customizing QuickDraw Operations" beginning on page 3-129.

The `grafProcs` field of a graphics port determines which low-level routines are called. If that field contains the value of `NIL`, the standard routines are called, so that all operations in that graphics port are done in the standard ways described in this chapter. You can set the `grafProcs` field to point to a record of pointers to your own routines. This record of pointers is defined by a data structure of type `QDProcs`, which is described on page 3-39.

To assist you in setting up a record, QuickDraw provides the `SetStdProcs` procedure, which is described on page 3-130. You can use the `SetStdProcs` procedure to get a `QDProcs` record with fields that point to the standard routines. You can reset the ones with which you are concerned. You can replace these low-level routines with your own, and then point to your modified `QDProcs` record in the `grafProcs` field of a `GrafPort` record to change basic QuickDraw's standard low-level behavior.

IMPORTANT

When modifying the low-level routines for a color graphics port, you must always use the `SetStdCProcs` procedure instead of `SetStdProcs`. ▲

The chapter "Pictures" in this book provides sample code and explanations for changing the standard low-level routines for reading and writing pictures.

QuickDraw Drawing Reference

This section describes the data structures, routines, and resources that QuickDraw provides to assist you in drawing lines and shapes onscreen.

"Data Structures" shows the Pascal data structures for the `Polygon`, `PenState`, `QDProcs`, and `Pattern` data types.

"Routines" describes QuickDraw routines for drawing lines, rectangles, rounded rectangles, ovals, arcs, wedges, polygons, and regions. "Routines" also describes routines for calculating, scaling, mapping, copying bits between, and otherwise manipulating these graphic entities.

"Resources" describes the pattern and pattern list resources, which your application can create to define its own bit patterns for drawing lines and shapes.

Data Structures

This section describes the `Polygon` record, the `PenState` record, the `QDProcs` record, and the `Pattern` record. Although this chapter describes routines for creating and manipulating rectangles and regions, the data structures you can use to define these entities are described in the chapter "Basic QuickDraw."

Your application typically does not create `Polygon` or `Pattern` records. Instead, you use the `OpenPoly` function (described on page 3-78) to create a polygon, and QuickDraw creates the necessary record. Although you can create a `Pattern` record in your program code, it is usually easier to create patterns using the pattern or pattern list resources, which are described beginning on page 3-140.

You need to use the `QDProcs` record only if you customize one or more of QuickDraw's low-level drawing routines. You can use the `SetStdProcs` procedure, described on page 3-130, to create a `QDProcs` record.

You can use a `PenState` record to save the location, size, pattern, and pattern mode of a graphics pen. The `GetPenState` procedure, described on page 3-43, automatically creates a pen state record. You can use the `SetPenState` procedure, described on page 3-43, to restore the values stored in a `PenState` record to the current graphics pen.

Polygon

After you use the `OpenPoly` function to create a polygon, QuickDraw begins collecting the line-drawing information you provide into a `Polygon` record, which is a data structure of type `Polygon`. The `OpenPoly` function returns a handle to the newly allocated `Polygon` record. Thereafter, your application normally refers to your new polygon by this handle, because QuickDraw routines such as `FramePoly` and `PaintPoly` expect a handle to a `Polygon` record as their first parameter.

A polygon is defined by a sequence of connected lines. A `Polygon` record consists of two fixed-length fields followed by a variable-length array of points: the starting point followed by each successive point to which a line is drawn.

```
Polygon =
   RECORD
      polySize:   Integer; {size in bytes}
      polyBBox:   Rect;    {bounding rectangle}
      polyPoints: ARRAY[0..0] OF Point;      {vertices of polygon}
   END;
```

Field descriptions

`polySize`	The size in bytes of this record.
`polyBBox`	The rectangle that bounds the polygon.
`polyPoints`	An array of points: the starting point followed by each successive point to which a line is drawn.

PenState

The `GetPenState` procedure (described on page 3-43) saves the location, size, pattern, and pattern mode of the graphics pen for the current graphics port in a `PenState` record, which is a data structure of type `PenState`. After changing the graphics pen as necessary, you can later restore these pen states with the `SetPenState` procedure (described on page 3-43).

Here is how a `PenState` record is defined:

```
TYPE PenState =
   RECORD
      pnLoc:   Point;      {pen location}
      pnSize:  Point;      {pen size}
      pnMode:  Integer;    {pen's pattern mode}
      pnPat:   Pattern;    {pen pattern}
   END;
```

Field descriptions

`pnLoc` For the current graphics port at the time the `GetPenState` procedure was called, the value of that graphics port's `pnLoc` field. This value is the point where QuickDraw begins drawing next. The location of the graphics pen is a point in the graphics port's coordinate system, not a pixel in a bit image. The upper-left corner of the pen is at the pen location; the graphics pen hangs below and to the right of this point.

`pnSize` For the current graphics port at the time the `GetPenState` procedure was called, the value of that graphics port's `pnSize` field. The graphics pen is rectangular in shape, and its width and height are specified by the values in the `pnSize` field. The default size is a 1-by-1 bit square; the width and height can range from 0 by 0 to 32,767 by 32,767. If either the pen width or the pen height is 0, the pen does not draw. Heights or widths of less than 0 are undefined.

`pnMode` The pattern mode—that is, for the current graphics port at the time the `GetPenState` procedure was called, the value of that graphics port's `pnMode` field. This value determines how the pen pattern is to affect what's already in the bit image when lines or shapes are drawn. When the graphics pen draws, QuickDraw first determines what bits in the bit image are affected, finds their corresponding bits in the pattern, and then transfers the bits from the pattern into the image according to this mode, which specifies one of eight Boolean transfer operations. The resulting bit is stored into its proper place in the bit image. The various pattern modes are described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8.

`pnPat` For the current graphics port at the time the `GetPenState` procedure was called, the pen pattern for that graphics port. This pattern determines how the bits under the graphics pen are affected when lines or shapes are drawn.

QDProcs

You need to use the `QDProcs` record, which is a data structure of type `QDProcs`, only if you customize one or more of QuickDraw's low-level drawing routines. You can use the `SetStdProcs` procedure, described on page 3-130, to create a `QDProcs` record.

The `QDProcs` record contains pointers to low-level drawing routines. QuickDraw's standard low-level drawing routines are described in "Customizing QuickDraw Operations" beginning on page 3-129. You can change the fields of this record to point to routines of your own devising.

```
TYPE QDProcsPtr = ^QDProcs;
QDProcs =
   RECORD
      textProc:   Ptr;  {text drawing}
      lineProc:   Ptr;  {line drawing}
      rectProc:   Ptr;  {rectangle drawing}
      rRectProc:  Ptr;  {roundRect drawing}
      ovalProc:   Ptr;  {oval drawing}
      arcProc:    Ptr;  {arc/wedge drawing}
      polyProc:   Ptr;  {polygon drawing}
      rgnProc:    Ptr;  {region drawing}
      bitsProc:   Ptr;  {bit transfer}
      commentProc:Ptr;  {picture comment processing}
      txMeasProc: Ptr;  {text width measurement}
      getPicProc: Ptr;  {picture retrieval}
      putPicProc: Ptr;  {picture saving}
    END;
```

Field descriptions

`textProc` A pointer to the low-level routine that draws text. The standard QuickDraw routine is the `StdText` procedure.

`lineProc` A pointer to the low-level routine that draws lines. The standard QuickDraw routine is the `StdLine` procedure.

`rectProc` A pointer to the low-level routine that draws rectangles. The standard QuickDraw routine is the `StdRect` procedure.

`rRectProc` A pointer to the low-level routine that draws rounded rectangles. The standard QuickDraw routine is the `StdRRect` procedure.

`ovalProc` A pointer to the low-level routine that draws ovals. The standard QuickDraw routine is the `StdOval` procedure.

`arcProc` A pointer to the low-level routine that draws arcs. The standard QuickDraw routine is the `StdArc` procedure.

`polyProc` A pointer to the low-level routine that draws polygons. The standard QuickDraw routine is the `StdPoly` procedure.

`rgnProc`	A pointer to the low-level routine that draws regions. The standard QuickDraw routine is the `StdRgn` procedure.
`bitsProc`	A pointer to the low-level routine that copies bitmaps. The standard QuickDraw routine is the `StdBits` procedure.
`commentProc`	A pointer to the low-level routine for processing a picture comment. The standard QuickDraw routine is the `StdComment` procedure.
`txMeasProc`	A pointer to the low-level routine for measuring text width. The standard QuickDraw routine is the `StdTxtMeas` function.
`getPicProc`	A pointer to the low-level routine for retrieving information from the definition of a picture. The standard QuickDraw routine is the `StdGetPic` procedure.
`putPicProc`	A pointer to the low-level routine for saving information as the definition of a picture. The standard QuickDraw routine is the `StdPutPic` procedure.

You can use the `SetStdProcs` procedure to set all the fields of the `QDProcs` record to point to QuickDraw's standard routines, and then reset the ones for which you have your own routines.

Pattern

Your application typically does not create `Pattern` records, which are data structures of type `Pattern`. Although you can create `Pattern` records in your program code, it is usually easier to create bit patterns using the pattern or pattern list resource, described beginning on page 3-140.

A bit pattern is a 64-bit image, organized as an 8-by-8 bit square, that defines a repeating design or tone. When a pattern is drawn, it's aligned so that adjacent areas of the same pattern in the same graphics port form a continuous, coordinated pattern. QuickDraw provides predefined patterns in global variables named `white`, `black`, `gray`, `ltGray`, and `dkGray`. A `Pattern` record is defined as follows:

```
TYPE  PatPtr      = ^Pattern;
      PatHandle   = ^PatPtr;
      Pattern     = PACKED ARRAY[0..7] OF 0..255;
```

The row width of a pattern is 1 byte.

Routines

This section describes QuickDraw's routines for drawing lines, rectangles, rounded rectangles, ovals, arcs, wedges, polygons, and regions. This section also describes routines for calculating, scaling, mapping, copying bits between, and otherwise manipulating these graphic entities.

These routines use your current graphics port as their drawing environment. You can use these routines to draw into basic graphics ports (described in the chapter "Basic QuickDraw") and color graphics ports (described in the chapter "Color QuickDraw").

See the chapter "Pictures" for descriptions of routines that create and draw pictures. See the chapter "Cursor Utilities" for information about drawing cursors. See the chapter "QuickDraw Text" in *Inside Macintosh: Text* for descriptions of QuickDraw's routines for drawing and manipulating text.

Managing the Graphics Pen

Every graphics port contains one, and only one, graphics pen with which to perform drawing operations. You use this metaphorical pen to draw lines, shapes, and text.

You can use the `HidePen` and `ShowPen` procedures to change the pen's visibility, the `PenSize` procedure to change its shape, the `PenPat` procedure to change its pattern, and the `PenMode` procedure to change its pattern mode. To determine the size, location, pattern, and pattern mode of the graphics pen, you can use the `GetPenState` procedure. If you need to temporarily change these characteristics, you can use the `SetPenState` procedure to restore the graphics pen to a state previously captured by `GetPenState`.

Upon the creation of a graphics port, QuickDraw assigns these initial values to the graphics pen: a size of (1,1), a pattern of all-black pixels, and the `patCopy` pattern mode. After changing any of these values, you can use the `PenNormal` procedure to return these initial values to the graphics pen.

These pen-manipulation routines use the local coordinate system of the current graphics port. Remember that each graphics port has its own pen, the state of which is stored in several fields of its `GrafPort` or `CGrafPort` record. If you draw in one graphics port, change to another, and return to the first, the pen for the first graphics port has the same state as when you left it. (The basic graphics port is described in the chapter "Basic QuickDraw," and the color graphics port is described in the chapter "Color QuickDraw.")

HidePen

To give the graphics pen invisible ink (which means that pen drawing doesn't show on the screen), use the `HidePen` procedure.

```
PROCEDURE HidePen;
```

DESCRIPTION

The `HidePen` procedure decrements the `pnVis` field of the current graphics port. The `pnVis` field is initialized to 0 by the `OpenPort` procedure. Whenever `pnVis` is negative, the pen doesn't draw on the screen. The `pnVis` field keeps track of the number of times the pen has been hidden to compensate for nested calls to the `HidePen` and `ShowPen` routines.

Every call to `HidePen` should be balanced by a subsequent call to `ShowPen`, which is described next.

The `HidePen` procedure is called by the `OpenRgn`, `OpenPicture`, and `OpenPoly` routines so that you can create regions, pictures, and polygons without drawing on the screen.

ShowPen

To change the ink of a graphics pen from invisible (which means that pen drawing doesn't show on the screen) to visible (so that pen drawing does appear on the screen), you can use the `ShowPen` procedure.

```
PROCEDURE ShowPen;
```

DESCRIPTION

The `ShowPen` procedure increments the `pnVis` field of the current graphics port. For 0 or positive values, pen drawing shows on the screen.

For example, if you have used the `HidePen` procedure to decrement the `pnVis` field from 0 to –1, you can use the `ShowPen` procedure to make its value 0 so that QuickDraw resumes drawing on the screen. Subsequent calls to `ShowPen` increment `pnVis` beyond 0, so every call to `ShowPen` should be balanced by a call to `HidePen`.

`ShowPen` is called by the procedures `CloseRgn` (described on page 3-89), `ClosePoly` (described on page 3-79), and `ClosePicture` (described in the chapter "Pictures" in this book).

GetPen

To determine the location of the graphics pen, use the `GetPen` procedure.

```
PROCEDURE GetPen (VAR pt: Point);
```

`pt` The graphics pen's current position in the current graphics port.

DESCRIPTION

In the `pt` parameter, the `GetPen` procedure returns the current pen position. The point returned is in the local coordinates of the current graphics port.

GetPenState

To determine the graphics pen's location, size, pattern, and pattern mode, you can use the `GetPenState` procedure.

```
PROCEDURE GetPenState (VAR pnState: PenState);
```

`pnState` A `PenState` record for holding information about the graphics pen.

DESCRIPTION

The `GetPenState` procedure saves the location, size, pattern, and pattern mode of the graphics pen for the current graphics port in a `PenState` record, which the `GetPenState` procedure returns in the `pnState` parameter.

After changing the graphics pen as necessary, you can later restore these pen states with the `SetPenState` procedure (described next).

SEE ALSO

The `PenState` record is described on page 3-37.

SetPenState

To restore the state of the graphics pen that was saved with the `GetPenState` procedure, use the `SetPenState` procedure.

```
PROCEDURE SetPenState (pnState: PenState);
```

`pnState` A `PenState` record previously created with the `GetPenState` procedure.

DESCRIPTION

The `SetPenState` procedure sets the graphics pen's location, size, pattern, and pattern mode in the current graphics port to the values stored in the `PenState` record that you specify in the `pnState` parameter.

SEE ALSO

The `PenState` record is described on page 3-37.

PenSize

To set the dimensions of the graphics pen in the current graphics port, use the `PenSize` procedure.

```
PROCEDURE PenSize (width,height: Integer);
```

`width` The pen width, as an integer from 0 to 32,767. If you set the value to 0, the pen does not draw. Values less than 0 are undefined.

`height` The pen height, as an integer from 0 to 32,767. If you set the value to 0, the pen does not draw. Values less than 0 are undefined.

DESCRIPTION

The `PenSize` procedure sets the width that you specify in the `width` parameter and the height that you specify in the `height` parameter for the graphics pen in the current graphics port. All subsequent calls to the `Line` and `LineTo` procedures and to the procedures that draw framed shapes in the current graphics port use the new pen dimensions.

You can get the current pen dimensions from the `pnSize` field of the current graphics port, where the width and height are stored as a `Point` record.

SEE ALSO

Listing 3-2 on page 3-20 illustrates how to use this procedure.

PenMode

To set the pattern mode of the graphics pen in the current graphics port, use the `PenMode` procedure.

```
PROCEDURE PenMode (mode: Integer);
```

`mode` The pattern mode. The following list shows the constants you can use—and the values they represent—for specifying the pattern mode.

```
CONST
   patCopy     = 8;  {where pattern pixel is black, apply }
                     { foreground color to destination pixel; }
                     { where pattern pixel is white, apply }
                     { background color to destination pixel}
   patOr       = 9;  {where pattern pixel is black, invert }
                     { destination pixel; where pattern }
                     { pixel is white, leave }
                     { destination pixel unaltered}
   patXor      = 10; {where pattern pixel is black, invert }
                     { destination pixel; where pattern }
                     { pixel is white, leave destination }
                     { pixel unaltered}
   patBic      = 11; {where pattern pixel is black, apply }
                     { background color to destination pixel; }
                     { where pattern pixel is white, leave }
                     { destination pixel unaltered}
   notPatCopy  = 12; {where pattern pixel is black, apply }
                     { background color to destination pixel; }
                     { where pattern pixel is white, apply }
                     { foreground color to destination pixel}
   notPatOr    = 13; {where pattern pixel is black, leave }
                     { destination pixel unaltered; where }
                     { pattern pixel is white, apply }
                     { foreground color to destination pixel}
   notPatXor   = 14; {where pattern pixel is black, }
                     { leave destination pixel unaltered; }
                     { where pattern pixel is white, }
                     { invert destination pixel}
   notPatBic   = 15; {where pattern pixel is black, }
                     { leave destination pixel unaltered; }
                     { where pattern pixel is white, apply }
                     { background color to destination pixel}
```

DESCRIPTION

Using the pattern mode you specify in the `mode` parameter, the `PenMode` procedure sets the manner in which the pattern of the graphics pen is transferred onto the bitmap (or pixel map) when you draw lines or shapes in the current graphics port. These actions are illustrated in Figure 3-4 on page 3-10.

If you specify a source mode (such as one used with the `CopyBits` procedure) instead of a pattern mode, no drawing is performed.

The current pattern mode is stored in the `pnMode` field of the current graphics port. The initial pattern mode value is `patCopy`, in which the pen pattern is copied directly to the bitmap.

To use highlighting, you can add this constant or its value to the source or pattern mode:

```
CONST
  hilite    = 50; {add to source or pattern mode for highlighting}
```

With highlighting, QuickDraw replaces the background color with the highlight color when your application draws or copies images between graphics ports. This has the visual effect of using a highlighting pen to select the object. (The global variable `HiliteRGB` is read from parameter RAM when the machine starts. Basic graphics ports use the color stored in the `HiliteRGB` global variable as the highlight color. Color graphics ports default to the `HiliteRGB` global variable, but can be overridden by the `HiliteColor` procedure, described in the chapter "Color QuickDraw.")

SPECIAL CONSIDERATIONS

When your application draws with a pixel pattern, Color QuickDraw ignores the pattern mode and simply transfers the pattern directly to the pixel map without regard to the foreground and background colors.

The results of inverting a pixel are predictable only with direct pixels or 1-bit pixel maps. For indexed pixels, Color QuickDraw performs the inversion on the pixel indexes, which means the results depend entirely on the contents of the color table (which is described in the chapter "Color QuickDraw"). The eight colors used in basic QuickDraw are stored in a color table represented by the global variable `QDColors`. To display those eight basic QuickDraw colors on an indexed device, Color QuickDraw uses the Color Manager to obtain indexes to the colors in the CLUT that best map to the colors in the `QDColors` color table. Because the index, not the color value, is inverted, the results are unpredictable.

SEE ALSO

Pattern modes are discussed in detail in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8 of this chapter and in "Boolean Transfer Modes With Color Pixels" beginning on page 4-32 in the chapter "Color QuickDraw."

PenPat

To set the bit pattern to be used by the graphics pen in the current graphics port, use the `PenPat` procedure.

```
PROCEDURE PenPat (pat: Pattern);
```

`pat` A bit pattern, as defined by a `Pattern` record.

DESCRIPTION

The `PenPat` procedure sets the graphics pen to use the bit pattern defined in the `Pattern` record that you specify in the `pat` parameter. (The standard patterns `white`, `black`, `gray`, `ltGray`, and `dkGray` are predefined; the initial bit pattern for the pen is `black`.) This pattern is stored in the `pnPat` field of a `GrafPort` record. The QuickDraw painting procedures (such as `PaintRect`) also use the pen's pattern when drawing a shape.

The `PenPat` procedure also sets a bit pattern for the graphics pen in a color graphics port. The `PenPat` procedure creates a handle, of type `PixPatHandle`, for the bit pattern and stores this handle in the `pnPixPat` field of the `CGrafPort` record. This pattern always uses the graphics port's current foreground and background colors.

SPECIAL CONSIDERATIONS

The `PenPat` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

The `Pattern` record is described on page 3-40. To define your own patterns, you typically create pattern or pattern list resources, which are described beginning on page 3-140.

The `CGrafPort` record is described in the chapter "Color QuickDraw." To set the graphics pen to use a multicolored pixel pattern in a color graphics port, use the `PenPixPat` procedure, which is described in the chapter "Color QuickDraw."

Listing 3-3 on page 3-21 illustrates how to use the `PenPat` procedure.

PenNormal

To set the size, pattern, and pattern mode of the graphics pen in the current graphics port to their initial values, use the `PenNormal` procedure.

```
PROCEDURE PenNormal;
```

DESCRIPTION

The `PenNormal` procedure restores the size, pattern, and pattern mode of the graphics pen in the current graphics port to their initial values: a size of 1 pixel by 1 pixel, a pattern mode of `patCopy`, and a pattern of `black`. The pen location does not change.

SPECIAL CONSIDERATIONS

The `PenNormal` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

Changing the Background Bit Pattern

Each graphics port has a background pattern that's used when an area is erased (such as by using the `EraseRect`, `EraseRoundRect`, `EraseArc`, `ErasePoly`, and `EraseRgn` procedures) and when pixels are scrolled out of an area (such as by using the `ScrollRect` procedure described in the chapter "Basic QuickDraw"). The background pattern is stored in the `bkPat` field of every basic graphics port. You can use the `BackPat` procedure to change the bit pattern used as the background color by the current graphics port (black and white or color).

BackPat

To change the bit pattern used as the background pattern by the current graphics port, use the `BackPat` procedure.

```
PROCEDURE BackPat (pat: Pattern);
```

`pat` A bit pattern, as defined by a `Pattern` record.

DESCRIPTION

The `BackPat` procedure sets the bit pattern defined in the `Pattern` record, which you specify in the `pat` parameter, to be the background pattern. (The standard bit patterns `white`, `black`, `gray`, `ltGray`, and `dkGray` are predefined; the initial background pattern for the graphics port is `white`.) This pattern is stored in the `bkPat` field of a `GrafPort` record.

The `BackPat` procedure also sets a bit pattern for the background color in a color graphics port. The `BackPat` procedure creates a handle, of type `PixPatHandle`, for the bit pattern and stores this handle in the `bkPixPat` field of the `CGrafPort` record. As in basic graphics ports, Color QuickDraw draws patterns in color graphics ports at the time of drawing, not at the time you use `PenPat` to set the pattern.

SPECIAL CONSIDERATIONS

The `BackPat` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

The `Pattern` record is described on page 3-40. To define your own patterns, you typically create pattern or pattern list resources, which are described beginning on page 3-140.

The `CGrafPort` record is described in the chapter "Color QuickDraw." To use a multicolored background pattern in a color graphics port, use the `BackPixPat` procedure, which is described in the chapter "Color QuickDraw."

Drawing Lines

A line is defined by two points: the current location of the graphics pen and its destination. You specify where to begin drawing a line by using the `MoveTo` or `Move` procedure to place the graphics pen at some point in the window's local coordinate system, and then using the `LineTo` or `Line` procedure to draw a line from there to another point.

MoveTo

To move the graphics pen to a particular location in the current graphics port, use the `MoveTo` procedure.

```
PROCEDURE MoveTo (h,v: Integer);
```

`h`	The horizontal coordinate of the graphics pen's new position.
`v`	The vertical coordinate of the graphics pen's new position.

DESCRIPTION

The `MoveTo` procedure changes the graphics pen's current location to the new horizontal coordinate you specify in the `h` parameter and the new vertical coordinate you specify in the `v` parameter. Specify the new location in the local coordinates of the current graphics port. The `MoveTo` procedure performs no drawing.

SEE ALSO

Listing 3-1 on page 3-18 illustrates how to use this procedure.

Move

To move the graphics pen a particular distance, use the `Move` procedure.

```
PROCEDURE Move (dh,dv: Integer);
```

`dh`	The horizontal distance of the graphics pen's movement.
`dv`	The vertical distance of the graphics pen's movement.

DESCRIPTION

The `Move` procedure moves the graphics pen from its current location in the current graphics port a horizontal distance that you specify in the `dh` parameter and a vertical distance that you specify in the `dv` parameter. The `Move` procedure calls

```
MoveTo(h+dh,v+dv)
```

where `(h,v)` is the graphics pen's current location in local coordinates. The `Move` procedure performs no drawing.

SEE ALSO

Listing 3-1 on page 3-18 illustrates how to use this procedure.

LineTo

To draw a line from the graphics pen's current location to a new location, use the `LineTo` procedure.

```
PROCEDURE LineTo (h,v: Integer);
```

`h`	The horizontal coordinate of the graphics pen's new location.
`v`	The vertical coordinate of the graphics pen's new location.

DESCRIPTION

The `LineTo` procedure draws a line from the graphics pen's current location in the current graphics port to the new location `(h,v)`, which you specify in the local coordinates of the current graphics port. If you are using `LineTo` to draw a region or polygon, its outline is infinitely thin and is not affected by the values of the `pnSize`, `pnMode`, or `pnPat` field of the graphics port.

SPECIAL CONSIDERATIONS

The `LineTo` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

Listing 3-1 on page 3-18 illustrates how to use this procedure.

Line

To draw a line a specified distance from the graphics pen's current location in the current graphics port, use the `Line` procedure.

```
PROCEDURE Line (dh,dv: Integer);
```

`dh`	The horizontal distance of the graphics pen's movement.
`dv`	The vertical distance of the graphics pen's movement.

DESCRIPTION

Starting at the current location of the graphics pen, the `Line` procedure draws a line the horizontal distance that you specify in the `dh` parameter and the vertical distance that you specify in the `dv` parameter. The `Line` procedure calls

```
LineTo(h+dh,v+dv)
```

where `(h,v)` is the current location in local coordinates. The pen location becomes the coordinates of the end of the line after the line is drawn. If you are using `Line` to draw a region or polygon, its outline is infinitely thin and is not affected by the values of the `pnSize`, `pnMode`, and `pnPat` fields of the graphics port.

SPECIAL CONSIDERATIONS

The `Line` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

Listing 3-1 on page 3-18 illustrates how to use this procedure.

Creating and Managing Rectangles

You can use a rectangle, which is defined by a `Rect` record, to specify locations and sizes for various graphics operations. (The `Rect` data type is described in the chapter "Basic QuickDraw.") You can use the `SetRect` procedure to create a rectangle, `OffsetRect` to move one, and `InsetRect` to shrink or expand one. You can determine whether two rectangles intersect with the `SectRect` procedure, whether a pixel is enclosed in a rectangle with the `PtInRect` procedure, whether two rectangles are equal with the `EqualRect` procedure, and whether a rectangle is an empty rectangle with the `EmptyRect` procedure. You can use the `UnionRect` procedure to calculate the smallest rectangle that encloses two other rectangles, `PtToAngle` to calculate the angle from the middle of a rectangle to a point, and `Pt2Rect` to determine the smallest rectangle that encloses two points.

If the points or rectangles supplied to these routines are defined in a graphics port other than your current graphics port, you must convert them to the local coordinate system of your current graphics port. You can accomplish this by using the `SetPort` procedure to change to the graphics port containing the points or rectangles, using the `LocalGlobal` procedure to convert their locations to global coordinates, using `SetPort` to return to your starting graphics port, and then using the `GlobalToLocal` procedure to convert the locations of points or rectangles to the local coordinates of your current graphics port. These procedures are described in the chapter "Basic QuickDraw."

SetRect

To assign coordinates to a rectangle, you can use the `SetRect` procedure.

```
PROCEDURE SetRect (VAR r: Rect; left,top,right,bottom: Integer);
```

`r`	The rectangle to set.
`left`	The horizontal coordinate of the new upper-left corner of the rectangle.
`top`	The vertical coordinate of the new upper-left corner of the rectangle.
`right`	The horizontal coordinate of the new lower-right corner of the rectangle.
`bottom`	The vertical coordinate of the new lower-right corner of the rectangle.

DESCRIPTION

The `SetRect` procedure assigns the coordinates you specify in the `left`, `top`, `right`, and `bottom` parameters to the rectangle that you specify in the `r` parameter. This procedure is provided to help you shorten your program text. If you want a more readable text, at the expense of source text length, you can instead assign integers (or points) directly into the fields of a `Rect` record.

SEE ALSO

Listing 3-4 on page 3-23 illustrates how to use this procedure. The data structure of type `Rect` is described in the chapter "Basic QuickDraw."

OffsetRect

To move a rectangle, use the `OffsetRect` procedure.

```
PROCEDURE OffsetRect (VAR r: Rect; dh,dv: Integer);
```

`r`	The rectangle to move.
`dh`	The horizontal distance to move the rectangle.
`dv`	The vertical distance to move the rectangle.

DESCRIPTION

The `OffsetRect` procedure moves the rectangle that you specify in the `r` parameter by adding the value you specify in the `dh` parameter to each of its horizontal coordinates and the value you specify in the `dv` parameter to each of its vertical coordinates. If the `dh` and `dv` parameters are positive, the movement is to the right and down; if either is negative, the corresponding movement is in the opposite direction. The rectangle retains its shape and size; it's merely moved on the coordinate plane. The movement doesn't affect the screen unless you subsequently call a routine to draw within the rectangle.

InsetRect

To shrink or expand a rectangle, use the `InsetRect` procedure.

```
PROCEDURE InsetRect (VAR r: Rect; dh,dv: Integer);
```

`r`	The rectangle to alter.
`dh`	The horizontal distance to move the left and right sides in toward or outward from the center of the rectangle.
`dv`	The vertical distance to move the top and bottom sides in toward or outward from the center of the rectangle.

DESCRIPTION

The `InsetRect` procedure shrinks or expands the rectangle that you specify in the `r` parameter: the left and right sides are moved in by the amount you specify in the `dh` parameter; the top and bottom are moved toward the center by the amount you specify in the `dv` parameter. If the value you pass in `dh` or `dv` is negative, the appropriate pair of sides is moved outward instead of inward. The effect is to alter the size by `2*dh` horizontally and `2*dv` vertically, with the rectangle remaining centered in the same place on the coordinate plane.

If the resulting width or height becomes less than 1, the rectangle is set to the empty rectangle (0,0,0,0).

SectRect

To determine whether two rectangles intersect, you can use the `SectRect` function.

```
FUNCTION SectRect (src1,src2: Rect; VAR dstRect: Rect): Boolean;
```

`src1`	The first of two rectangles to test for intersection.
`src2`	The second of two rectangles to test for intersection.
`dstRect`	The rectangle marking the intersection of the first two rectangles.

DESCRIPTION

The `SectRect` function calculates the rectangle that delineates the intersection of the two rectangles you specify in the `src1` and `src2` parameters. The `SectRect` function returns the area of intersection in the `dstRect` parameter and a function result of `TRUE` if they intersect or `FALSE` if they don't. Rectangles that touch at a line or a point are not considered intersecting, because their intersection rectangle (actually, in this case, an intersection line or point) doesn't enclose any pixels in the bit image.

If the rectangles don't intersect, the destination rectangle is set to (0,0,0,0). The `SectRect` procedure works correctly even if one of the source rectangles is also the destination.

UnionRect

To calculate the smallest rectangle that encloses two rectangles, use the `UnionRect` procedure.

```
PROCEDURE UnionRect (src1,src2: Rect; VAR dstRect: Rect);
```

`src1`	The first of two rectangles to enclose.
`src2`	The second of two rectangles to enclose.
`dstRect`	The rectangle that encloses them.

DESCRIPTION

The `UnionRect` procedure returns in the `dstRect` parameter the smallest rectangle that encloses both of the rectangles you specify in the `src1` and `src2` parameters. One of the source rectangles may also be the destination.

PtInRect

To determine whether a pixel below is enclosed in a rectangle, use the `PtInRect` function.

```
FUNCTION PtInRect (pt: Point; r: Rect): Boolean;
```

`pt`	The point to test.
`r`	The rectangle to test.

DESCRIPTION

The `PtInRect` function determines whether the pixel below and to the right of the point you specify in the `pt` parameter is enclosed in the rectangle that you specify in the `Rect` parameter. The `PtInRect` function returns `TRUE` if it is, `FALSE` if it is not.

Pt2Rect

To determine the smallest rectangle that encloses two given points, use the `Pt2Rect` procedure.

```
PROCEDURE Pt2Rect (pt1,pt2: Point; VAR dstRect: Rect);
```

`pt1`	The first of two points to enclose.
`pt2`	The second of two points to enclose.
`dstRect`	The smallest rectangle that can enclose them.

DESCRIPTION

The `Pt2Rect` procedure returns in the `dstRect` parameter the smallest rectangle that encloses the two points you specify in the `pt1` and `pt2` parameters.

PtToAngle

To calculate an angle between a vertical line pointing straight up from the center of a rectangle and a line from the center to a given point, use the `PtToAngle` procedure.

```
PROCEDURE PtToAngle (r: Rect; pt: Point; VAR angle: Integer);
```

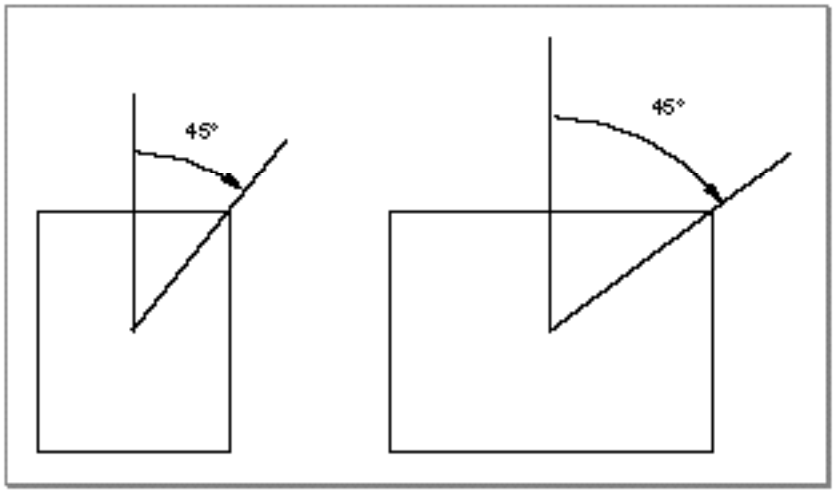
`r`	The rectangle to examine.
`pt`	The point to which an angle is to be calculated.
`angle`	The resulting angle.

DESCRIPTION

The `PtToAngle` procedure returns in the `angle` parameter the angle between a vertical line (pointing straight up from the center of the rectangle that you specify in the `r` parameter) and a line from the center of that rectangle to a point (which you specify in the `pt` parameter).

The result returned in the `angle` parameter is specified in degrees from 0 to 359, measured clockwise from 12 o'clock, with 90° at 3 o'clock, 180° at 6 o'clock, and 270° at 9 o'clock. Other angles are measured relative to the rectangle. If the line to the given point goes through the upper-right corner of the rectangle, the angle returned is 45°, even if the rectangle isn't square; if it goes through the lower-right corner, the angle is 135°, and so on, as shown in Figure 3-18.

Figure 3-18 Forty-five-degree angles as returned by the `PtToAngle` procedure



The angle returned might be used as input to one of the procedures that manipulate arcs and wedges, as described in "Drawing Arcs and Wedges" beginning on page 3-71.

EqualRect

To determine whether two rectangles are equal, you can use the `EqualRect` function.

```
FUNCTION EqualRect (rect1,rect2: Rect): Boolean;
```

`rect1` The first of two rectangles to compare.

`rect2` The second of two rectangles to compare.

DESCRIPTION

The `EqualRect` function compares the rectangles you specify in the `rect1` and `rect2` parameters and returns `TRUE` if they're equal, `FALSE` if they're not.

EmptyRect

To determine whether a rectangle is an empty rectangle, use the `EmptyRect` function.

```
FUNCTION EmptyRect (r: Rect): Boolean;
```

`r` The rectangle to examine.

DESCRIPTION

The `EmptyRect` function returns `TRUE` if the rectangle that you specify in the `r` parameter is an empty rectangle, `FALSE` if it is not. A rectangle is considered empty if the bottom coordinate is less than or equal to the top coordinate or if the right coordinate is less than or equal to the left.

Drawing Rectangles

A rectangle is defined by a data structure of type `Rect`, in which you specify two points (for the upper-left and lower-right corners of the rectangle) or four boundary coordinates (one for each side of the rectangle). After defining a rectangle (such as by using the `SetRect` procedure), you can use the `FrameRect` procedure to outline it with the size, pattern, and pattern mode of the graphics pen.

You can use the `PaintRect` procedure to draw a rectangle's interior with the pattern of the graphics pen, using the pattern mode of the graphics pen.

Using the `FillRect` procedure, you can draw a rectangle's interior with any pattern you specify. The procedure transfers the pattern with the `patCopy` pattern mode, which directly copies your requested pattern into the shape.

You can use the `EraseRect` procedure to erase a rectangle; this procedure fills the rectangle's interior with the background pattern for the current graphics port. Making the shape blend into the background pattern of the graphics port effectively erases the shape. For example, you can use `EraseRect` to erase the port rectangle for a window before redrawing into the window.

You can use the `InvertRect` procedure to invert a rectangle; this procedure reverses the colors of all pixels within the rectangle's boundary. On a black-and-white monitor, this changes all black pixels in the shape to white, and changes all white pixels to black. Although this procedure operates on color pixels in color graphics ports, the results are predictable only with direct pixels or 1-bit pixel maps.

FrameRect

To draw an outline inside a rectangle, use the `FrameRect` procedure.

```
PROCEDURE FrameRect (r: Rect);
```

`r` The rectangle to frame.

DESCRIPTION

Using the pattern, pattern mode, and size of the graphics pen for the current graphics port, the `FrameRect` procedure draws an outline just inside the rectangle that you specify in the `r` parameter. The outline is as wide as the pen width and as tall as the pen height. The pen location does not change.

If a region is open and being formed, the outside outline of the new rectangle is mathematically added to the region's boundary.

SPECIAL CONSIDERATIONS

The `FrameRect` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

Listing 3-4 on page 3-23 illustrates how to use this procedure.

PaintRect

To paint a rectangle with the graphics pen's pattern and pattern mode, use the `PaintRect` procedure.

```
PROCEDURE PaintRect (r: Rect);
```

`r` The rectangle to paint.

DESCRIPTION

The `PaintRect` procedure draws the interior of the rectangle that you specify in the `r` parameter with the pen pattern for the current graphics port, according to the pattern mode for the current graphics port. The pen location does not change.

SPECIAL CONSIDERATIONS

The `PaintRect` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

Listing 3-5 on page 3-24 illustrates how to use this procedure.

You can use the `FillRect` procedure, described next, to draw the interior of a rectangle with a pen pattern different from that for the current graphics port.

FillRect

To fill a rectangle with any available bit pattern, use the `FillRect` procedure.

```
PROCEDURE FillRect (r: Rect; pat: Pattern);
```

`r` The rectangle to fill.

`pat` The bit pattern to use for the fill. Figure 3-3 on page 3-7 illustrates the default fill patterns and the constants you can use to represent them.

DESCRIPTION

Using the `patCopy` pattern mode, the `FillRect` procedure draws the interior of the rectangle that you specify in the `r` parameter with the pattern defined in the `Pattern` record that you specify in the `pat` parameter. This procedure leaves the pen location unchanged.

SPECIAL CONSIDERATIONS

The `FillRect` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

Listing 3-5 on page 3-24 illustrates how to use this procedure.

The `patCopy` pattern mode is described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8. The `Pattern` record is described on page 3-40. You can use the `GetPattern` and `GetIndPattern` routines, described on page 3-126 and page 3-127, respectively, to get a pattern stored in a resource.

You can use the `PaintRect` procedure, described in the previous section, to draw the interior of a rectangle with the pen pattern for the current graphics port. To fill a rectangle with a pixel pattern, use the `FillCRect` procedure, which is described in the chapter "Color QuickDraw."

EraseRect

To erase a rectangle, use the `EraseRect` procedure.

```
PROCEDURE EraseRect (r: Rect);
```

`r` The rectangle to erase.

DESCRIPTION

Using the `patCopy` pattern mode, the `EraseRect` procedure draws the interior of the rectangle that you specify in the `r` parameter with the background pattern for the current graphics port. This effectively erases the rectangle specified in the `r` parameter. For example, you can use `EraseRect` to erase the port rectangle for a window before redrawing into the window.

This procedure leaves the location of the graphics pen unchanged.

SPECIAL CONSIDERATIONS

The `EraseRect` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

Listing 6-2 on page 6-10 in the chapter "Offscreen Graphics Worlds" in this book illustrates how to use `EraseRect` to initialize an offscreen pixel map. The `patCopy` pattern mode is described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8.

InvertRect

To invert the pixels enclosed by a rectangle, use the `InvertRect` procedure.

```
PROCEDURE InvertRect (r: Rect);
```

`r` The rectangle whose enclosed pixels are to be inverted.

DESCRIPTION

The `InvertRect` procedure inverts the pixels enclosed by the rectangle that you specify in the `r` parameter. Every white pixel becomes black and every black pixel becomes white. The pen location does not change.

SPECIAL CONSIDERATIONS

The `InvertRect` procedure was designed for 1-bit images in basic graphics ports. This procedure operates on color pixels in color graphics ports, but the results are predictable only with direct pixels or 1-bit pixel maps. For indexed pixels, Color QuickDraw performs the inversion on the pixel indexes, which means the results depend entirely on the contents of the CLUT (which is described in the chapter "Color QuickDraw"). The eight colors used in basic QuickDraw are stored in a color table represented by the global variable `QDColors`. To display those eight basic QuickDraw colors on an indexed device, Color QuickDraw uses the Color Manager to obtain indexes to the colors in the CLUT that best map to the colors in the `QDColors` color table. Because the index, not the color value, is inverted, the results are unpredictable.

Inversion works better for direct pixels. Inverting a pure green, for example, that has red, green, and blue component values of $0000, $FFFF, and $0000 results in magenta, which has component values of $FFFF, $0000, and $FFFF.

The `InvertRect` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

Drawing Rounded Rectangles

As with rectangles, QuickDraw provides routines with which you can frame, paint, fill, erase, and invert rounded rectangles. Rounded rectangles are rectangles with rounded corners defined by the width and height of the ovals forming their corners.

You can use the `FrameRoundRect` procedure to draw an outline of a rounded rectangle with the size, pattern, and pattern mode of the graphics pen. You can use the `PaintRoundRect` procedure to draw a rounded rectangle's interior with the pattern of the graphics pen, using the pattern mode of the graphics pen.

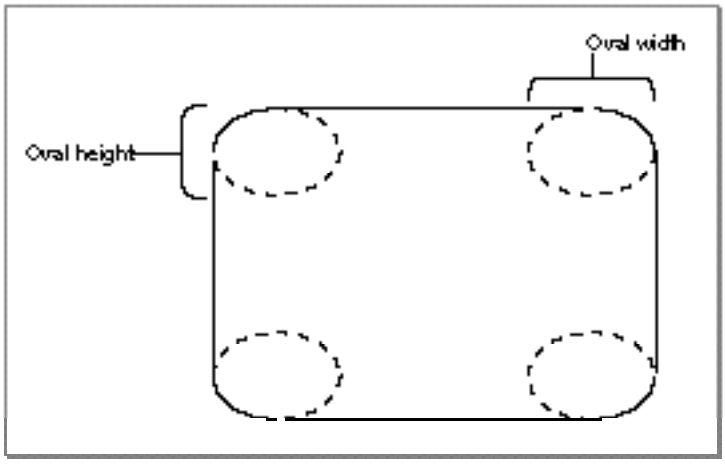
Using the `FillRoundRect` procedure, you can draw a rounded rectangle's interior with any pattern you specify. The procedure transfers the pattern with the `patCopy` pattern mode, which directly copies your requested pattern into the shape.

You can use the `EraseRoundRect` procedure to erase a rounded rectangle; this procedure fills the rectangle's interior with the background pattern for the current graphics port.

You can use the `InvertRoundRect` procedure to invert a rounded rectangle; this procedure reverses the colors of all pixels within the rounded rectangle. Although this procedure operates on color pixels in color graphics ports, the results are predictable only with 1-bit and direct color pixels.

When using these procedures, you specify a rectangle, which is defined by a data structure of type `Rect`. You must also specify the width and height of the ovals that describe the curvature of the rounded corners, as shown in Figure 3-19.

Figure 3-19 Oval width and height in rounded rectangles



FrameRoundRect

To draw an outline inside a rounded rectangle, use the `FrameRoundRect` procedure.

```
PROCEDURE FrameRoundRect (r: Rect; ovalWidth,ovalHeight: Integer);
```

`r` The rectangle that defines the rounded rectangle's boundaries.

`ovalWidth` The width of the oval defining the rounded corner.

`ovalHeight` The height of the oval defining the rounded corner.

DESCRIPTION

Using the pattern, pattern mode, and size of the graphics pen for the current graphics port, the `FrameRoundRect` procedure draws an outline just inside the rounded rectangle bounded by the rectangle that you specify in the `r` parameter. The outline is as wide as the pen width and as tall as the pen height. The pen location does not change.

Use the `ovalWidth` and `ovalHeight` parameters to specify the diameters of curvature for the corners of the rounded rectangle.

If a region is open and being formed, the outside outline of the new rounded rectangle is mathematically added to the region's boundary.

SPECIAL CONSIDERATIONS

The `FrameRoundRect` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

PaintRoundRect

To paint a rounded rectangle with the graphics pen's pattern and pattern mode, use the `PaintRoundRect` procedure.

```
PROCEDURE PaintRoundRect (r: Rect; ovalWidth,ovalHeight: Integer);
```

`r` The rectangle that defines the rounded rectangle's boundaries.

`ovalWidth` The width of the oval defining the rounded corner.

`ovalHeight` The height of the oval defining the rounded corner.

DESCRIPTION

Using the pattern and pattern mode of the graphics pen for the current graphics port, the `PaintRoundRect` procedure draws the interior of the rounded rectangle bounded by the rectangle that you specify in the `r` parameter. Use the `ovalWidth` and `ovalHeight` parameters to specify the diameters of curvature for the corners of the rounded rectangle.

The pen location does not change.

SPECIAL CONSIDERATIONS

The `PaintRoundRect` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

You can use the `FillRoundRect` procedure, described next, to draw the interior of a rounded rectangle with a pen pattern different from that for the current graphics port.

FillRoundRect

To fill a rounded rectangle with any available bit pattern, use the `FillRoundRect` procedure.

```
PROCEDURE FillRoundRect (r: Rect; ovalWidth,ovalHeight: Integer;
                         pat: Pattern);
```

`r`	The rectangle that defines the rounded rectangle's boundaries.
`ovalWidth`	The width of the oval defining the rounded corner.
`ovalHeight`	The height of the oval defining the rounded corner.
`pat`	The bit pattern to use for the fill. Figure 3-3 on page 3-7 illustrates the default fill patterns and the constants you can use to represent them.

DESCRIPTION

Using the `patCopy` pattern mode, the `FillRoundRect` procedure draws the interior of the rounded rectangle bounded by the rectangle that you specify in the `r` parameter with the bit pattern defined in the `Pattern` record that you specify in the `pat` parameter. Use the `ovalWidth` and `ovalHeight` parameters to specify the diameters of curvature for the corners. The pen location does not change.

SPECIAL CONSIDERATIONS

The `FillRoundRect` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

You can use the `GetPattern` and `GetIndPattern` routines, described on page 3-126 and page 3-127, respectively, to get a pattern stored in a resource. The `patCopy` pattern mode is described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8. The `Pattern` record is described on page 3-40.

You can use the `PaintRoundRect` procedure, described in the previous section, to draw the interior of a rounded rectangle with the pen pattern for the current graphics port. To fill a rounded rectangle with a pixel pattern, use the `FillCRoundRect` procedure, which is described in the chapter "Color QuickDraw."

EraseRoundRect

To erase a rounded rectangle, use the `EraseRoundRect` procedure.

```
PROCEDURE EraseRoundRect (r: Rect; ovalWidth,ovalHeight: Integer);
```

`r`	The rectangle that defines the rounded rectangle's boundaries.
`ovalWidth`	The width of the oval defining the rounded corner.
`ovalHeight`	The height of the oval defining the rounded corner.

DESCRIPTION

Using the `patCopy` pattern mode, the `EraseRoundRect` procedure draws the interior of the rounded rectangle bounded by the rectangle that you specify in the `r` parameter with the background pattern of the current graphics port. This effectively erases the rounded rectangle. Use the `ovalWidth` and `ovalHeight` parameters to specify the diameters of curvature for the corners of the rounded rectangle.

This procedure leaves the location of the graphics pen unchanged.

SPECIAL CONSIDERATIONS

The `EraseRoundRect` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

The `patCopy` pattern mode is described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8.

InvertRoundRect

To invert the pixels enclosed by a rounded rectangle, use the `InvertRoundRect` procedure.

```
PROCEDURE InvertRoundRect (r: Rect;
                           ovalWidth,
                           ovalHeight: Integer);
```

`r`	The rectangle that defines the rounded rectangle's boundaries.
`ovalWidth`	The width of the oval defining the rounded corner.
`ovalHeight`	The height of the oval defining the rounded corner.

DESCRIPTION

The `InvertRoundRect` procedure inverts the pixels enclosed by the rounded rectangle bounded by the rectangle that you specify in the `r` parameter. Every white pixel becomes black and every black pixel becomes white. The `ovalWidth` and `ovalHeight` parameters specify the diameters of curvature for the corners. The pen location does not change.

SPECIAL CONSIDERATIONS

The `InvertRoundRect` procedure was designed for 1-bit images in basic graphics ports. This procedure operates on color pixels in color graphics ports, but the results are predictable only with direct devices or 1-bit pixel maps. For indexed pixels, Color QuickDraw performs the inversion on the pixel indexes, which means the results depend entirely on the contents of the CLUT (which is described in the chapter "Color QuickDraw"). The eight colors used in basic QuickDraw are stored in a color table represented by the global variable `QDColors`. To display those eight basic QuickDraw colors on an indexed device, Color QuickDraw uses the Color Manager to obtain indexes to the colors in the CLUT that best map to the colors in the `QDColors` color table. Because the index, not the color value, is inverted, the results are unpredictable.

Inversion works better for direct pixels. Inverting a pure green, for example, that has red, green, and blue component values of $0000, $FFFF, and $0000 results in magenta, which has component values of $FFFF, $0000, and $FFFF.

The `InvertRoundRect` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

Drawing Ovals

An oval is a circular or elliptical shape defined by the bounding rectangle that encloses it. You can use the `FrameOval` procedure to draw its outline, or the `PaintOval` or `FillOval` procedure to draw its interior with a pattern. You can use the `EraseOval` procedure to erase an oval, and you can use the `InvertOval` procedure to reverse the colors of all pixels within the oval. (Although this procedure operates on color pixels in color graphics ports, the results of `InvertOval` are predictable only with 1-bit or direct color pixels.)

FrameOval

To draw an outline inside an oval, use the `FrameOval` procedure.

```
PROCEDURE FrameOval (r: Rect);
```

`r` The rectangle that defines the oval's boundary.

DESCRIPTION

Using the pattern, pattern mode, and size of the graphics pen for the current graphics port, the `FrameOval` procedure draws an outline just inside the oval with the bounding rectangle that you specify in the `r` parameter. The outline is as wide as the pen width and as tall as the pen height. The pen location does not change.

If a region is open and being formed, the outside outline of the new oval is mathematically added to the region's boundary.

SPECIAL CONSIDERATIONS

The `FrameOval` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

Listing 3-6 on page 3-25 illustrates how to use this procedure.

PaintOval

To paint an oval with the graphics pen's pattern and pattern mode, use the `PaintOval` procedure.

```
PROCEDURE PaintOval (r: Rect);
```

`r` The rectangle that defines the oval's boundary.

DESCRIPTION

Using the pen pattern and pattern mode for the current graphics port, the `PaintOval` procedure draws the interior of an oval just inside the bounding rectangle that you specify in the `r` parameter. The pen location does not change.

SPECIAL CONSIDERATIONS

The `PaintOval` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

You can use the `FillOval` procedure, described next, to draw the interior of an oval with a pen pattern different from that for the current graphics port.

FillOval

To fill an oval with any available bit pattern, use the `FillOval` procedure.

```
PROCEDURE FillOval (r: Rect; pat: Pattern);
```

`r` The rectangle that defines the oval's boundaries.

`pat` The bit pattern to use for the fill. Figure 3-3 on page 3-7 illustrates the default fill patterns and the constants you can use to represent them.

DESCRIPTION

Using the `patCopy` pattern mode and the bit pattern defined in the `Pattern` record that you specify in the `pat` parameter, the `FillOval` procedure draws the interior of an oval just inside the bounding rectangle that you specify in the `r` parameter. The pen location does not change.

SPECIAL CONSIDERATIONS

The `FillOval` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

You can use the `GetPattern` and `GetIndPattern` routines, described on page 3-126 and page 3-127, respecively, to get a pattern stored in a resource. The `patCopy` pattern mode is described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8. The `Pattern` record is described on page 3-40.

You can use the `PaintOval` procedure, described in the previous section, to draw the interior of an oval with the pen pattern for the current graphics port. To fill an oval with a pixel pattern, use the `FillCOval` procedure, which is described in the chapter "Color QuickDraw."

EraseOval

To erase an oval, use the `EraseOval` procedure.

```
PROCEDURE EraseOval (r: Rect);
```

`r` The rectangle that defines the oval's boundary.

DESCRIPTION

Using the background pattern for the current graphics port and the `patCopy` pattern mode, the `EraseOval` procedure draws the interior of an oval just inside the bounding rectangle that you specify in the `r` parameter. This effectively erases the oval bounded by the specified rectangle.

This procedure leaves the location of the graphics pen unchanged.

SPECIAL CONSIDERATIONS

The `EraseOval` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

The `patCopy` pattern mode is described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8.

InvertOval

To invert the pixels enclosed by an oval, use the `InvertOval` procedure.

```
PROCEDURE InvertOval (r: Rect);
```

`r` The rectangle that defines the oval's boundary.

DESCRIPTION

The `InvertOval` procedure inverts the pixels enclosed by an oval just inside the bounding rectangle that you specify in the `r` parameter. Every white pixel becomes black and every black pixel becomes white. The pen location does not change.

SPECIAL CONSIDERATIONS

The `InvertOval` procedure was designed for 1-bit images in basic graphics ports. This procedure operates on color pixels in color graphics ports, but the results are predictable only with direct devices or 1-bit pixel maps. For indexed pixels, Color QuickDraw performs the inversion on the pixel indexes, which means the results depend entirely on the contents of the CLUT (which is described in the chapter "Color QuickDraw"). The eight colors used in basic QuickDraw are stored in a color table represented by the global variable `QDColors`. To display those eight basic QuickDraw colors on an indexed device, Color QuickDraw uses the Color Manager to obtain indexes to the colors in the CLUT that best map to the colors in the `QDColors` color table. Because the index, not the color value, is inverted, the results are unpredictable.

Inversion works better for direct pixels. Inverting a pure green, for example, that has red, green, and blue component values of $0000, $FFFF, and $0000 results in magenta, which has component values of $FFFF, $0000, and $FFFF.

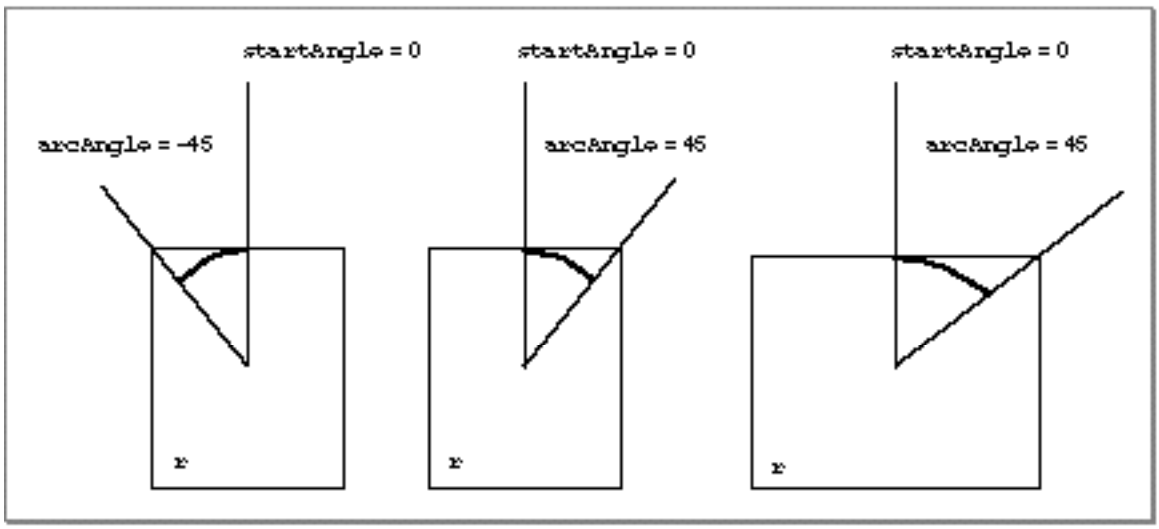
The `InvertOval` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

Drawing Arcs and Wedges

An arc is defined as a portion of an oval's circumference bounded by a pair of radii. A wedge is a pie-shaped segment bounded by a pair of radii, and it extends from the center of the oval to the circumference. You use the `FrameArc` procedure to draw an arc, and you use the `PaintArc` or `FillArc` procedure to draw a wedge. Using the `EraseArc` procedure, you can erase a wedge, and, using `InvertArc`, you can reverse the colors of all pixels within a wedge. (Although this procedure operates on color pixels in color graphics ports, the results of `InvertArc` are predictable only with 1-bit and direct color pixels.)

These procedures take three parameters: a rectangle that defines an oval's boundaries, an angle indicating the start of the arc (the variable `startAngle`), and an angle indicating the arc's extent (the variable `arcAngle`). For the angle parameters, 0° indicates a vertical line straight up from the center of the oval. Positive values indicate angles in the clockwise direction from this vertical line, and negative values indicate angles in the counterclockwise direction, as shown in Figure 3-20.

Figure 3-20 Using angles to define the radii for arcs and wedges



FrameArc

To draw an arc of the oval that fits inside a rectangle, use the `FrameArc` procedure.

```
PROCEDURE FrameArc (r: Rect; startAngle,arcAngle: Integer);
```

`r` The rectangle that defines an oval's boundaries.

`startAngle`
The angle indicating the start of the arc.

`arcAngle` The angle indicating the arc's extent.

DESCRIPTION

Using the pattern, pattern mode, and size of the graphics pen for the current graphics port, the `FrameArc` procedure draws an arc of the oval bounded by the rectangle that you specify in the `r` parameter. Use the `startAngle` parameter to specify where the arc begins as modulo 360. Use the `arcAngle` parameter to specify how many degrees the arc covers. Specify whether the angles are in positive or negative degrees; a positive angle goes clockwise, while a negative angle goes counterclockwise. Zero degrees is at 12 o'clock high, 90° (or –270°) is at 3 o'clock, 180° (or –180°) is at 6 o'clock, and 270° (or –90°) is at 9 o'clock. Measure other angles relative to the bounding rectangle.

A line from the center of the rectangle through its upper-right corner is at 45°, even if the rectangle isn't square; a line through the lower-right corner is at 135°, and so on, as shown in Figure 3-20.

The arc is as wide as the pen width and as tall as the pen height. The pen location does not change.

SPECIAL CONSIDERATIONS

The `FrameArc` procedure differs from other QuickDraw procedures that frame shapes in that the arc is not mathematically added to the boundary of a region that's open and being formed.

The `FrameArc` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

Listing 3-7 on page 3-26 illustrates how to use this procedure.

PaintArc

To paint a wedge of the oval that fits inside a rectangle with the graphics pen's pattern and pattern mode, use the `PaintArc` procedure.

```
PROCEDURE PaintArc (r: Rect; startAngle,arcAngle: Integer);
```

`r` The rectangle that defines an oval's boundaries.

`startAngle`
The angle indicating the start of the arc.

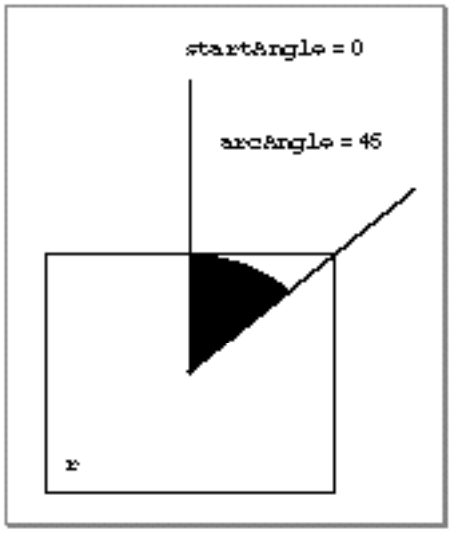
`arcAngle` The angle indicating the arc's extent.

DESCRIPTION

Using the pen pattern and pattern mode of the current graphics port, the `PaintArc` procedure draws a wedge of the oval bounded by the rectangle that you specify in the `r` parameter. As in the `FrameArc` procedure described in the previous section and illustrated in Figure 3-21, use the `startAngle` and `arcAngle` parameters to define the arc of the wedge.

The pen location does not change.

Figure 3-21 Using `PaintArc` to paint a 45° angle



SPECIAL CONSIDERATIONS

The `PaintArc` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

Listing 3-7 on page 3-26 illustrates how to use this procedure.

You can use the `FillArc` procedure, described next, to draw a wedge with a pattern different from that specified in the `pnPat` field of the current graphics port.

FillArc

To fill a wedge with any available bit pattern, use the `FillArc` procedure.

```
PROCEDURE FillArc (r: Rect; startAngle,arcAngle: Integer;
                   pat: Pattern);
```

`r`	The rectangle that defines an oval's boundaries.
`startAngle`	The angle indicating the start of the arc.
`arcAngle`	The angle indicating the arc's extent.
`pat`	The bit pattern to use for the fill. Figure 3-3 on page 3-7 illustrates the default fill patterns and the constants you can use to represent them.

DESCRIPTION

Using the `patCopy` pattern mode and the pattern defined in the `Pattern` record that you specify in the `pat` parameter, the `FillArc` procedure draws a wedge of the oval bounded by the rectangle that you specify in the `r` parameter. As in the `FrameArc` procedure described on page 3-72 and as illustrated in Figure 3-21, use the `startAngle` and `arcAngle` parameters to define the arc of the wedge.

This procedure leaves the location of the graphics pen unchanged.

SPECIAL CONSIDERATIONS

The `FillArc` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

You can use the `GetPattern` and `GetIndPattern` routines, described on page 3-126 and page 3-127, respectively, to get a pattern stored in a resource. The `patCopy` pattern mode is described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8. The `Pattern` record is described on page 3-40.

You can use the `PaintArc` procedure, described in the previous section, to draw a wedge with the pen pattern for the current graphics port. To fill a wedge with a pixel pattern, use the `FillCArc` procedure, which is described in the chapter "Color QuickDraw."

EraseArc

To erase a wedge, use the `EraseArc` procedure.

```
PROCEDURE EraseArc (r: Rect; startAngle,arcAngle: Integer);
```

`r` The rectangle that defines an oval's boundaries.

`startAngle` The angle indicating the start of the arc.

`arcAngle` The angle indicating the arc's extent.

DESCRIPTION

Using the `patCopy` pattern mode, the `EraseArc` procedure draws a wedge of the oval bounded by the rectangle that you specify in the `r` parameter with the background pattern for the current graphics port. As in the `FrameArc` procedure described on page 3-72 and as illustrated in Figure 3-21, use the `startAngle` and `arcAngle` parameters to define the arc of the wedge.

This procedure leaves the location of the graphics pen unchanged.

SPECIAL CONSIDERATIONS

The `EraseArc` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

The `patCopy` pattern mode is described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8.

InvertArc

To invert the pixels of a wedge, use the `InvertArc` procedure.

```
PROCEDURE InvertArc (r: Rect; startAngle,arcAngle: Integer);
```

`r` The rectangle that defines an oval's boundaries.

`startAngle`
The angle indicating the start of the arc.

`arcAngle` The angle indicating the arc's extent.

DESCRIPTION

The `InvertArc` procedure inverts the pixels enclosed by a wedge of the oval bounded by the rectangle that you specify in the `r` parameter. Every white pixel becomes black and every black pixel becomes white. As in the `FrameArc` procedure described on page 3-72 and as illustrated in Figure 3-21, use the `startAngle` and `arcAngle` parameters to define the arc of the wedge.

This procedure leaves the location of the graphics pen unchanged.

SPECIAL CONSIDERATIONS

The `InvertArc` procedure was designed for 1-bit images in basic graphics ports. This procedure operates on color pixels in color graphics ports, but the results are predictable only with direct devices or 1-bit pixel maps. For indexed pixels, Color QuickDraw performs the inversion on the pixel indexes, which means the results depend entirely on the contents of the CLUT (which is described in the chapter "Color QuickDraw"). The eight colors used in basic QuickDraw are stored in a color table represented by the global variable `QDColors`. To display those eight basic QuickDraw colors on an indexed device, Color QuickDraw uses the Color Manager to obtain indexes to the colors in the CLUT that best map to the colors in the `QDColors` color table. Because the index, not the color value, is inverted, the results are unpredictable.

Inversion works better for direct pixels. Inverting a pure green, for example, that has red, green, and blue component values of $0000, $FFFF, and $0000 results in magenta, which has component values of $FFFF, $0000, and $FFFF.

The `InvertArc` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

Creating and Managing Polygons

A polygon is defined by a sequence of connected lines. To create a polygon, you first call the `OpenPoly` function and then some number of `LineTo` procedures to draw lines from the first vertex of the polygon to the second, from the second to the third, and so on, until you've drawn a line to the last vertex. You then use the `ClosePoly` procedure, which completes the figure by drawing a connecting line from the last vertex back to the first.

After you use the `OpenPoly` function to create a polygon, QuickDraw begins collecting the line-drawing information you provide into a `Polygon` record. The `OpenPoly` function returns a handle to the newly allocated `Polygon` record.

After defining a polygon in this way, you can draw it with the `FramePoly`, `PaintPoly`, and `FillPoly` procedures. You can move it by using the `OffSetPoly` procedure. When you are finished using the polygon, use the `KillPoly` procedure to release its memory. When using the `ClosePoly`, `OffsetPoly`, and `KillPoly` procedures, you refer to a polygon by the handle returned by `OpenPoly` when you first created the polygon.

Note
If, while your application draws a polygon, it exceeds available stack space in Color QuickDraw, the `QDError` function (described in the chapter "Color QuickDraw" in this book) returns the result code –144. ◆

OpenPoly

To begin defining a polygon, use the `OpenPoly` function.

```
FUNCTION OpenPoly:  PolyHandle;
```

DESCRIPTION

The `OpenPoly` function returns a handle to a new polygon and starts saving lines for processing as a polygon definition. While a polygon is open, all calls to the `Line` and `LineTo` procedures affect the outline of the polygon. Only the line endpoints affect the polygon definition; the pattern mode, pattern, and size do not affect it. The `OpenPoly` function calls the `HidePen` procedure, so no drawing occurs on the screen while the polygon is open (unless you call the `ShowPen` procedure just after calling `OpenPoly`, or you called `ShowPen` previously without balancing it by a call to `HidePen`).

A polygon should consist of a sequence of connected lines. The `OpenPoly` function stores the definition for a polygon in a `Polygon` record.

When a polygon is open, the current graphics port's `polySave` field contains a handle to information related to the polygon definition. If you want to temporarily disable the polygon definition, you can save the current value of this field, set the field to `NIL`, and later restore the saved value to resume the polygon definition.

Even though the onscreen presentation of a polygon is clipped, the definition of a polygon is not; you can define a polygon anywhere on the coordinate plane.

When you are finished calling the line-drawing routines that define your polygon, use the `ClosePoly` procedure, described next.

SPECIAL CONSIDERATIONS

Do not call `OpenPoly` while a region or another polygon is already open.

Polygons are limited to 64 KB. You can determine the polygon size while it's being formed by calling the Memory Manager function `GetHandleSize`, which is described in *Inside Macintosh: Memory*.

The `OpenPoly` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

SEE ALSO

Listing 3-10 on page 3-30 illustrates how to use this function to create a triangle. The `Polygon` record is described on page 3-37.

ClosePoly

To complete the collection of lines that defines your polygon, use the `ClosePoly` procedure.

```
PROCEDURE ClosePoly;
```

DESCRIPTION

The `ClosePoly` procedure stops collecting line-drawing commands for the currently open polygon and computes the `polyBBox` field of the `Polygon` record. You should call `ClosePoly` only once for every call to the `OpenPoly` function.

The `ClosePoly` procedure uses the `ShowPen` procedure, balancing the call to the `HidePen` procedure made by the `OpenPoly` function.

SPECIAL CONSIDERATIONS

The `ClosePoly` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

Listing 3-10 on page 3-30 illustrates how to use this procedure when creating a triangle.

OffsetPoly

To move a polygon, use the `OffsetPoly` procedure.

```
PROCEDURE OffsetPoly (poly: PolyHandle; dh,dv: Integer);
```

`poly`	A handle to a polygon to move.
`dh`	The horizontal distance to move the polygon.
`dv`	The vertical distance to move the polygon.

DESCRIPTION

The `OffsetPoly` procedure moves the polygon whose handle you pass in the `poly` parameter by adding the value you specify in the `dh` parameter to the horizontal coordinates of its points, and by adding the value you specify in the `dv` parameter to the vertical coordinates of all points of its region boundary. If the values of `dh` and `dv` are positive, the movement is to the right and down; if either is negative, the corresponding movement is in the opposite direction. The region retains its size and shape. This doesn't affect the screen unless you subsequently call a routine to draw the region.

Note
`OffsetPoly` is an especially efficient operation, because the data defining a polygon is stored relative to the first point of the polygon and so isn't actually changed by `OffsetPoly`. ◆

KillPoly

To release the memory occupied by a polygon, use the `KillPoly` procedure.

```
PROCEDURE KillPoly (poly: PolyHandle);
```

`poly`	A handle to the polygon to dispose of.

DESCRIPTION

The `KillPoly` procedure releases the memory used by the polygon whose handle you pass in the `poly` parameter. Use `KillPoly` only when you're completely through with a polygon.

SPECIAL CONSIDERATIONS

The `KillPoly` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

Listing 3-10 on page 3-30 illustrates how to use this procedure.

Drawing Polygons

After defining a polygon by using the `OpenPoly` function, a number of line-drawing procedures, and the `ClosePoly` procedure, you can draw the polygon's outline with the `FramePoly` procedure. You can draw its interior with the `PaintPoly` and `FillPoly` procedures. You can erase its interior by using the `ErasePoly` procedure, and you can use the `InvertPoly` procedure to reverse the colors of the pixels within it. In all of these procedures, you refer to a polygon by the handle returned by `OpenPoly` when you first created the polygon.

Four of these procedures—`PaintPoly`, `ErasePoly`, `InvertPoly`, and `FillPoly`—temporarily convert the polygon into a region to perform their operations. The amount of memory required for this temporary region may be far greater than the amount required by the polygon alone.

You can estimate the size of this region by scaling down the polygon with the `MapPoly` procedure (described on page 3-108), converting the polygon into a region, checking the region's size with the Memory Manager function `GetHandleSize`, and multiplying that value by the factor by which you scaled the polygon.

▲ **WARNING**
The results of these graphics operations are undefined whenever any horizontal or vertical line drawn through the polygon would intersect the polygon's outline more than 50 times. ▲

FramePoly

To draw the outline of a polygon, use the `FramePoly` procedure.

```
PROCEDURE FramePoly (poly:  PolyHandle);
```

`poly` A handle to the polygon to draw.

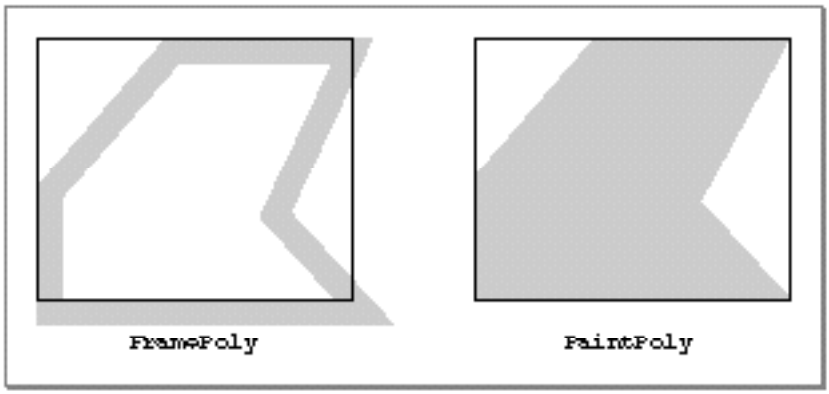
DESCRIPTION

Using the current graphics port's pen pattern, pattern mode, and size, the `FramePoly` procedure plays back the line-drawing commands that define the polygon whose handle you pass in the `poly` parameter.

The graphics pen hangs below and to the right of each point on the boundary of the polygon. Thus, the drawn polygon extends beyond the right and bottom edges of the polygon's bounding rectangle (which is stored in the `polyBBox` field of the `Polygon` record) by the pen width and pen height, respectively. All other graphics

operations, such as painting a polygon with the `PaintPoly` procedure, occur strictly within the boundary of the polygon, as illustrated in Figure 3-22.

Figure 3-22 Framing and painting polygons



If a polygon is open and being formed, `FramePoly` affects the outline of the polygon just as if the line-drawing routines themselves had been called. If a region is open and being formed, the outside outline of the polygon being framed is mathematically added to the region's boundary.

SPECIAL CONSIDERATIONS

The `FramePoly` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

PaintPoly

To paint a polygon with the graphics pen's pattern and pattern mode, use the `PaintPoly` procedure.

```
PROCEDURE PaintPoly (poly: PolyHandle);
```

`poly` A handle to the polygon to paint.

DESCRIPTION

Using the pen pattern and pattern mode for the current graphics port, the `PaintPoly` procedure draws the interior of a polygon whose handle you pass in the `poly` parameter. The pen location does not change.

SPECIAL CONSIDERATIONS

Do not create a height or width for the polygon greater than 32,767 pixels, or `PaintPoly` will crash.

The `PaintPoly` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

You can use the `FillPoly` procedure, described next, to draw the interior of a polygon with a pattern different from that specified in the `pnPat` field of the current graphics port.

FillPoly

To fill a polygon with any available bit pattern, use the `FillPoly` procedure.

```
PROCEDURE FillPoly (poly: PolyHandle; pat: Pattern);
```

`poly`	A handle to the polygon to fill.
`pat`	The bit pattern to use for the fill. Figure 3-3 on page 3-7 illustrates the default fill patterns and the constants you can use to represent them.

DESCRIPTION

Using the `patCopy` pattern mode, the `FillPoly` procedure draws the interior of the polygon whose handle you pass in the `poly` parameter with the pattern defined in the `Pattern` record that you specify in the `pat` parameter.

This procedure leaves the location of the graphics pen unchanged.

SPECIAL CONSIDERATIONS

The `FillPoly` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

Listing 3-10 on page 3-30 illustrates how to use this procedure to fill a triangle.

You can use the `GetPattern` and `GetIndPattern` routines, described on page 3-126 and page 3-127, respectively, to get a pattern stored in a resource. The `patCopy` pattern mode is described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8. The `Pattern` record is described on page 3-40.

You can use the `PaintPoly` procedure, described in the previous section, to draw the interior of a polygon with the pen pattern for the current graphics port. To fill a polygon with a pixel pattern, use the `FillCPoly` procedure, which is described in the chapter "Color QuickDraw."

ErasePoly

To erase a polygon, use the `ErasePoly` procedure.

```
PROCEDURE ErasePoly (poly: PolyHandle);
```

`poly` A handle to the polygon to erase.

DESCRIPTION

Using the `patCopy` pattern mode, the `ErasePoly` procedure draws the interior of the polygon whose handle you pass in the `poly` parameter with the background pattern for the current graphics port.

This procedure leaves the location of the graphics pen unchanged.

SPECIAL CONSIDERATIONS

The `ErasePoly` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

The `patCopy` pattern mode is described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8.

InvertPoly

To invert the pixels enclosed by a polygon, use the `InvertPoly` procedure.

```
PROCEDURE InvertPoly (poly: PolyHandle);
```

`poly` A handle to a polygon, the pixels of which you want to invert.

DESCRIPTION

The `InvertPoly` procedure inverts the pixels enclosed by the polygon whose handle you pass in the `poly` parameter. Every white pixel becomes black and every black pixel becomes white.

This procedure leaves the location of the graphics pen unchanged.

SPECIAL CONSIDERATIONS

The `InvertPoly` procedure was designed for 1-bit images in basic graphics ports. This procedure operates on color pixels in color graphics ports, but the results are predictable only with 1-bit or direct pixels. For indexed pixels, Color QuickDraw performs the inversion on the pixel indexes, which means the results depend entirely on the contents of the CLUT (which is described in the chapter "Color QuickDraw"). The eight colors used in basic QuickDraw are stored in a color table represented by the global variable `QDColors`. To display those eight basic QuickDraw colors on an indexed device, Color QuickDraw uses the Color Manager to obtain indexes to the colors in the CLUT that best map to the colors in the `QDColors` color table. Because the index, not the color value, is inverted, the results are unpredictable.

Inversion works better for direct pixels. Inverting a pure green, for example, that has red, green, and blue component values of $0000, $FFFF, and $0000 results in magenta, which has component values of $FFFF, $0000, and $FFFF.

The `InvertPoly` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

Creating and Managing Regions

To define a region, you can use any set of lines or shapes, including other regions, so long as the region's outline consists of one or more closed loops. To begin defining a region, you must use the `NewRgn` function to allocate space for it, and then call the `OpenRgn` procedure. You can then use any QuickDraw routines to construct the outline of the region. When you are finished constructing the region, use the `CloseRgn` procedure.

The NewRgn function returns a handle to the newly allocated Region record. After you use the OpenRgn procedure, QuickDraw begins collecting the drawing information you provide into this Region record. (The Region record is described in the chapter "Basic QuickDraw.")

After defining a region in this way, you can display it with the FrameRgn, PaintRgn, and FillRgn procedures. When you are finished using the region, use the DisposeRgn procedure to release its memory.

You can use the SetEmptyRgn procedure to set a region to be empty, SetRectRgn to change it into a rectangle, OffsetRgn to move it, InsetRgn to shrink or expand it, PtInRgn to determine whether a pixel lies within it, RectInRgn to determine whether a rectangle intersects it, EmptyRgn to determine whether it is an empty region, and CopyRgn to make a copy of it. You can use the RectRgn procedure to make a region out of a rectangle. You can use the SectRgn procedure to calculate the intersection of two regions, UnionRgn to calculate the union of two regions, DiffRgn to subtract one region from another, XorRgn to calculate the difference between the union and the intersection of two regions, and EqualRgn to determine whether two regions have identical sizes, shapes, and locations.

When using these procedures, you refer to a region by the handle returned by NewRgn when you first allocated memory for the region.

▲ **WARNING**

Ensure that the memory for a region is valid before calling these routines to manipulate that region; if there isn't sufficient memory, the system may crash. Regions are limited to 32 KB in size in basic QuickDraw and 64 KB in Color QuickDraw. Before defining a region, you can use the Memory Manager function MaxMem to determine whether the memory for the region is valid. You can determine the current size of an existing region by calling the Memory Manager function GetHandleSize. (Both MaxMem and GetHandleSize are described in *Inside Macintosh: Memory*.) When you record drawing operations in an open region, the resulting region description may overflow the 32 KB or 64 KB limit. Should this happen in Color QuickDraw, the QDError function (described in the chapter "Color QuickDraw" in this book) returns the result code regionTooBigError. ▲

If the points or rectangles supplied to these routines are defined in a graphics port other than your current graphics port, you must convert them to the local coordinate system of your current graphics port. You can accomplish this by using the SetPort procedure to change to the graphics port containing the points or rectangles, using the LocalGlobal procedure to convert their locations to global coordinates, using SetPort to return to your starting graphics port, and then using the GlobalToLocal procedure to convert the locations of points or rectangles to the local coordinates of your current graphics port. These procedures are described in the chapter "Basic QuickDraw."

NewRgn

To begin creating a new region, use the `NewRgn` function.

```
FUNCTION NewRgn: RgnHandle;
```

DESCRIPTION

The `NewRgn` function allocates space for a new, variable-size region; initializes it to the empty region defined by the rectangle (0,0,0,0); and returns a handle to the new region. This is the only function that creates a new region; other routines merely alter the size or shape of existing regions.

To begin defining a region, use the `OpenRgn` procedure, described next.

SPECIAL CONSIDERATIONS

The `NewRgn` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

Use the Memory Manager function `MaxMem` (described in *Inside Macintosh: Memory*) to determine whether the memory for the region is valid before using `NewRgn`.

SEE ALSO

Listing 3-8 on page 3-28 and Listing 3-9 on page 3-29 illustrate how to use this function.

OpenRgn

To begin defining a region, use the `OpenRgn` procedure.

```
PROCEDURE OpenRgn;
```

DESCRIPTION

The `OpenRgn` procedure allocates temporary memory to start saving lines and framed shapes for processing as a region definition. Call `OpenRgn` only after initializing a region with the `NewRgn` function.

The `NewRgn` function stores the definition for a region in a `Region` record.

While a region is open, all calls to `Line`, `LineTo`, and the procedures that draw framed shapes (except arcs) affect the outline of the region. Only the line endpoints and shape boundaries affect the region definition—the pattern mode, pattern, and size do not affect it.

When you are finished defining the region, call the `CloseRgn` procedure.

The OpenRgn procedure calls HidePen, so no drawing occurs on the screen while the region is open (unless you call ShowPen just after OpenRgn, or you called ShowPen previously without balancing it by a call to HidePen). Since the pen hangs below and to the right of the pen location, drawing lines with even the smallest pen changes pixels that lie outside the region you define.

The outline of a region is mathematically defined and infinitely thin, and it separates the bit or pixel image into two groups of pixels: those within the region and those outside it.

A region should consist of one or more closed loops. Each framed shape itself constitutes a loop. Any lines drawn with the Line or LineTo procedure should connect with each other or with a framed shape. Even if the onscreen presentation of a region is clipped, the definition of a region is not; you can define a region anywhere on the coordinate plane with complete disregard for the location of various graphics port entities on that plane.

When a region is open, the current graphics port's rgnSave field contains a handle to information related to the region definition. If you want to temporarily disable the collection of lines and shapes, you can save the current value of this field, set the field to NIL, and later restore the saved value to resume the region definition. Also, calling SetPort while a region is being formed discontinues formation of the region until another call to SetPort resets the region's original graphics port.

SPECIAL CONSIDERATIONS

Regions are limited to 32 KB in size in basic QuickDraw and 64 KB in Color QuickDraw. You can determine the current size of an existing region by calling the Memory Manager function GetHandleSize (described in *Inside Macintosh: Memory*). When you record drawing operations in an open region, the resulting region description may overflow the 32 KB or 64 KB limit. Should this happen in Color QuickDraw, the QDError function (described in the chapter "Color QuickDraw" in this book) returns the result code regionTooBigError.

Do not call OpenRgn while another region or a polygon is already open. When you are finished constructing the region, use the CloseRgn procedure, which is described next.

The OpenRgn procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

Listing 3-8 on page 3-28 illustrates how to use this procedure. The Region record is described in the chapter "Basic QuickDraw."

CloseRgn

To organize a collection of lines and shapes into a region definition, use the `CloseRgn` procedure.

```
PROCEDURE CloseRgn (dstRgn: rgnHandle);
```

`dstRgn` The handle to the region to close.

DESCRIPTION

The `CloseRgn` procedure stops the collection of lines and framed shapes, organizes them into a region definition, and saves the result in the region whose handle you pass in the `dstRgn` parameter. The handle you pass in the `dstRgn` parameter should be a region handle returned by the `NewRgn` function.

The `CloseRgn` procedure does not create the destination region; you must have already allocated space for it by using the `OpenRgn` procedure. The `CloseRgn` procedure calls the `ShowPen` procedure, balancing the call to the `HidePen` procedure made by `OpenRgn`.

When you no longer need the memory occupied by the region, use the `DisposeRgn` procedure, described next.

SPECIAL CONSIDERATIONS

Regions are limited to 32 KB in size in basic QuickDraw and 64 KB in Color QuickDraw. When you record drawing operations in an open region, the resulting region description may overflow this limit. Should this happen in Color QuickDraw, the `QDError` function (described in the chapter "Color QuickDraw" in this book) returns the result code `regionTooBigError`. Since the resulting region is potentially corrupt, the `CloseRgn` procedure returns an empty region if it detects `QDError` has returned `regionTooBigError`.

The `CloseRgn` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

Listing 3-8 on page 3-28 illustrates how to use this procedure.

DisposeRgn

To release the memory occupied by a region, use the `DisposeRgn` procedure.

```
PROCEDURE DisposeRgn (rgn: RgnHandle);
```

`rgn` A handle to the region to dispose.

DESCRIPTION

The `DisposeRgn` procedure releases the memory occupied by the region whose handle you pass in the `rgn` parameter.

Use `DisposeRgn` only after you're completely through with a region.

SPECIAL CONSIDERATIONS

The `DisposeRgn` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

Listing 3-8 on page 3-28 and Listing 3-9 on page 3-29 illustrate how to use this procedure.

CopyRgn

To make a copy of a region, use the `CopyRgn` procedure.

```
PROCEDURE CopyRgn (srcRgn,dstRgn: RgnHandle);
```

`srcRgn` A handle to the region to copy.

`dstRgn` A handle to the region to receive the copy.

DESCRIPTION

The `CopyRgn` procedure copies the mathematical structure of the region whose handle you pass in the `srcRgn` parameter into the region whose handle you pass in the `dstRgn` parameter; that is, `CopyRgn` makes a duplicate copy of `srcRgn`. When calling `CopyRgn`, pass handles that have been returned by the `NewRgn` function in the `srcRgn` and `dstRgn` parameters.

Once this is done, the region indicated by `srcRgn` may be altered (or even disposed of) without affecting the region indicated by `dstRgn`. The `CopyRgn` procedure does not create the destination region; space must already have been allocated for it by using the `NewRgn` function.

SPECIAL CONSIDERATIONS

The `CopyRgn` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SetEmptyRgn

To set an existing region to be empty, use the `SetEmptyRgn` procedure.

```
PROCEDURE SetEmptyRgn (rgn: RgnHandle);
```

`rgn` A handle to the region to be made empty.

DESCRIPTION

The `SetEmptyRgn` procedure destroys the previous structure of the region whose handle you pass in the `rgn` parameter; it then sets the new structure to the empty region defined by the rectangle (0,0,0,0).

SPECIAL CONSIDERATIONS

The `SetEmptyRgn` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SetRectRgn

To change the structure of an existing region to that of a rectangle, you can use the `SetRectRgn` procedure.

```
PROCEDURE SetRectRgn (rgn: RgnHandle;
                      left,top,right,bottom: Integer);
```

`rgn` A handle to the region to restructure as a rectangle.

`left` The horizontal coordinate of the upper-left corner of the rectangle to set as the new region.

`top` The vertical coordinate of the upper-left corner of the rectangle to set as the new region.

`right` The horizontal coordinate of the lower-right corner of the rectangle to set as the new region.

`bottom` The vertical coordinate of the lower-right corner of the rectangle to set as the new region.

DESCRIPTION

The `SetRectRgn` procedure destroys the previous structure of the region whose handle you pass in the `rgn` parameter, and it then sets the new structure to the rectangle that you specify in the `left`, `top`, `right`, and `bottom` parameters. If you specify an empty rectangle (that is, `right`<=`left` or `bottom`<=`top`), the `SetRectRgn` procedure sets the region to the empty region defined by the rectangle (0,0,0,0).

As an alternative to the `SetRectRgn` procedure, you can change the structure of an existing region to that of a rectangle by using the `RectRgn` procedure, which accepts as a parameter a rectangle instead of four coordinates. The `RectRgn` procedure is described next.

SPECIAL CONSIDERATIONS

The `SetRectRgn` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

RectRgn

To change the structure of an existing region to that of a rectangle, you can use the `RectRgn` procedure.

```
PROCEDURE RectRgn (rgn: RgnHandle; r: Rect);
```

`rgn`	A handle to the region to restructure as a rectangle.
`r`	The rectangle structure to use.

DESCRIPTION

The `RectRgn` procedure destroys the previous structure of the `SetRectRgn` procedure, and it then sets the new structure to a rectangle that you specify in the `r` parameter.

As an alternative to the `RectRgn` procedure, you can use the `SetRectRgn` procedure, which accepts as parameters four coordinates instead of a rectangle.

SPECIAL CONSIDERATIONS

The `RectRgn` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

OffsetRgn

To move a region, use the `OffsetRgn` procedure.

```
PROCEDURE OffsetRgn (rgn: RgnHandle; dh,dv: Integer);
```

`rgn`	A handle to the region to move.
`dh`	The horizontal distance to move the region.
`dv`	The vertical distance to move the region.

DESCRIPTION

The `OffsetRgn` procedure moves the region whose handle you pass in the `rgn` parameter by adding the value you specify in the `dh` parameter to the horizontal coordinates of all points of its region boundary, and by adding the value you specify in the `dv` parameter to the vertical coordinates of all points of its region boundary. If the values of `dh` and `dv` are positive, the movement is to the right and down; if either is negative, the corresponding movement is in the opposite direction. The region retains its size and shape. This doesn't affect the screen unless you subsequently call a routine to draw the region.

The `OffsetRgn` procedure is an especially efficient operation, because most of the data defining a region is stored relative to the `rgnBBox` field in its `Region` record and so isn't actually changed by `OffsetRgn`.

InsetRgn

To shrink or expand a region, use the `InsetRgn` procedure.

```
PROCEDURE InsetRgn (rgn: RgnHandle; dh,dv: Integer);
```

`rgn`	A handle to the region to alter.
`dh`	The horizontal distance to move points on the left and right boundaries in toward or outward from the center.
`dv`	The vertical distance to move points on the top and bottom boundaries in toward or outward from the center.

DESCRIPTION

The `InsetRgn` procedure moves all points on the region boundary of the region whose handle you pass in the `rgn` parameter inward by the vertical distance that you specify in the `dv` parameter and by the horizontal distance that you specify in the `dh` parameter. If you specify negative values for `dh` or `dv`, the `InsetRgn` procedure moves the points outward in that direction.

The InsetRgn procedure leaves the region's center at the same position, but moves the outline in (for positive values of dh and dv) or out (for negative values of dh and dv). Using InsetRgn on a rectangular region has the same effect as using the InsetRect procedure.

SPECIAL CONSIDERATIONS

The InsetRgn procedure temporarily uses heap space that's twice the size of the original region.

The InsetRgn procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SectRgn

To calculate the intersection of two regions, use the SectRgn procedure.

```
PROCEDURE SectRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);
```

srcRgnA	A handle to the first of two regions whose intersection is to be determined.
srcRgnB	A handle to the second of two regions whose intersection is to be determined.
dstRgn	A handle to the region to receive the intersection area.

DESCRIPTION

The SectRgn procedure calculates the intersection of the two regions whose handles you pass in the srcRgnA and srcRgnB parameters, and it places the intersection in the region whose handle you pass in the dstRgn parameter. If the regions do not intersect, or one of the regions is empty, SectRgn sets the destination to the empty region defined by the rectangle (0,0,0,0).

The SectRgn procedure does not create a destination region; you must have already allocated memory for it by using the NewRgn function.

The destination region may be one of the source regions, if desired.

SPECIAL CONSIDERATIONS

The `SectRgn` procedure may temporarily use heap space that's twice the size of the two input regions.

The `SectRgn` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

UnionRgn

To calculate the union of two regions, use the `UnionRgn` procedure.

```
PROCEDURE UnionRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);
```

`srcRgnA`	A handle to the first of two regions whose union is to be determined.
`srcRgnB`	A handle to the second of two regions whose union is to be determined.
`dstRgn`	A handle to the region to hold the resulting union area.

DESCRIPTION

The `UnionRgn` procedure calculates the union of the two regions whose handles you pass in the `srcRgnA` and `srcRgnB` parameters, and it places the union in the region whose handle you pass in the `dstRgn` parameter. If both regions are empty, `UnionRgn` sets the destination to the empty region defined by the rectangle (0,0,0,0).

The `UnionRgn` procedure does not create the destination region; you must have already allocated memory for it by using the `NewRgn` function.

The destination region may be one of the source regions, if desired.

SPECIAL CONSIDERATIONS

The `UnionRgn` procedure may temporarily use heap space that's twice the size of the two input regions.

The `UnionRgn` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

DiffRgn

To subtract one region from another, use the `DiffRgn` procedure.

```
PROCEDURE DiffRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);
```

`srcRgnA` A handle to the region to subtract from.

`srcRgnB` A handle to the region to subtract.

`dstRgn` A handle to the region to hold the resulting area.

DESCRIPTION

The `DiffRgn` procedure subtracts the region whose handle you pass in the `srcRgnB` parameter from the region whose handle you pass in the `srcRgnA` parameter and places the difference in the region whose handle you pass in the `dstRgn` parameter. If the first source region is empty, `DiffRgn` sets the destination to the empty region defined by the rectangle (0,0,0,0).

The `DiffRgn` procedure does not create the destination region; you must have already allocated memory for it by using the `NewRgn` function. The destination region may be one of the source regions, if desired.

SPECIAL CONSIDERATIONS

The `DiffRgn` procedure may temporarily use heap space that's twice the size of the two input regions.

XorRgn

To calculate the difference between the union and the intersection of two regions, use the `XorRgn` procedure.

```
PROCEDURE XorRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);
```

`srcRgnA` A handle to the first of two regions to compare.

`srcRgnB` A handle to the second of two regions to compare.

`dstRgn` A handle to the region to hold the result.

DESCRIPTION

The `XorRgn` procedure calculates the difference between the union and the intersection of the regions whose handles you pass in the `srcRgnA` and `srcRgnB` parameters and places the result in the region whose handle you pass in the `dstRgn` parameter.

This does not create the destination region; you must have already allocated memory for it by using the `NewRgn` function.

If the regions are coincident, `XorRgn` sets the destination region to the empty region defined by the rectangle (0,0,0,0).

SPECIAL CONSIDERATIONS

The `XorRgn` procedure may temporarily use heap space that's twice the size of the two input regions.

The `XorRgn` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

PtInRgn

To determine whether a pixel is within a region, use the `PtInRgn` function.

```
FUNCTION PtInRgn (pt: Point; rgn: RgnHandle): Boolean;
```

`pt`	The point whose pixel is to be checked.
`rgn`	A handle to the region to test.

DESCRIPTION

The `PtInRgn` function checks whether the pixel below and to the right of the point you specify in the `pt` parameter is within the region whose handle you pass in the `rgn` parameter. The `PtInRgn` function returns `TRUE` if so or `FALSE` if not.

RectInRgn

To determine whether a rectangle intersects a region, use the `RectInRgn` function.

```
FUNCTION RectInRgn (r: Rect; rgn: RgnHandle): Boolean;
```

`r`	The rectangle to check for intersection.
`rgn`	A handle to the region to check.

DESCRIPTION

The `RectInRgn` function checks whether the rectangle specified in the `r` parameter intersects the region whose handle you pass in the `rgn` parameter. The `RectInRgn` function returns `TRUE` if the intersection encloses at least 1 bit or `FALSE` if it does not.

SPECIAL CONSIDERATIONS

The `RectInRgn` function sometimes returns `TRUE` when the rectangle merely intersects the region's bounding rectangle. If you need to know exactly whether a given rectangle intersects the actual region, you can use the `RectRgn` procedure (described on page 3-92) to set the rectangle to a region, and call `SectRgn` (described on page 3-94) to see whether the two regions intersect. If the result of `SectRgn` is an empty region, then the rectangle doesn't intersect the region.

EqualRgn

To determine whether two regions have identical sizes, shapes, and locations, use the `EqualRgn` function.

```
FUNCTION EqualRgn (rgnA,rgnB: RgnHandle): Boolean;
```

`srcRgnA`	A handle to the first of two regions to compare.
`srcRgnB`	A handle to the second of two regions to compare.

DESCRIPTION

The `EqualRgn` function compares the two regions whose handles you pass in the `rgnA` and `rgnB` parameters and returns `TRUE` if they're equal or `FALSE` if they're not.

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

EmptyRgn

To determine whether a region is empty, use the `EmptyRgn` function.

```
FUNCTION EmptyRgn (rgn: RgnHandle): Boolean;
```

`rgn` A handle to the region to test for emptiness.

DESCRIPTION

The `EmptyRgn` function returns `TRUE` if the region whose handle you pass in the `rgn` parameter is an empty region or `FALSE` if it is not.

SEE ALSO

The `EmptyRgn` function does not create an empty region. To create an empty region, you can perform any of the following operations:

- use the `NewRgn` function (described on page 3-87)
- pass the handle to an empty region to the `CopyRgn` procedure (described on page 3-90)
- pass an empty rectangle to either the `SetRectRgn` procedure (described on page 3-91) or the `RectRgn` procedure (described on page 3-92)
- call the `CloseRgn` procedure (described on page 3-89) without a previous call to the `OpenRgn` procedure
- call `CloseRgn` without performing any drawing after calling `OpenRgn`
- pass an empty region to the `OffsetRgn` procedure (described on page 3-93)
- pass an empty region or too large an inset to the `InsetRgn` procedure (described on page 3-93)
- pass two nonintersecting regions to the `SectRgn` procedure (described on page 3-94)
- pass two empty regions to the `UnionRgn` procedure (described on page 3-95)
- pass two identical or nonintersecting regions to the `DiffRgn` (described on page 3-96) or `XorRgn` (described on page 3-96) procedure

Drawing Regions

After defining a region by using the `NewRgn` function and `OpenRgn` procedure, a number of drawing procedures, and the `CloseRgn` procedure, you can draw the region's outline with the `FrameRgn` procedure. You can draw its interior with the `PaintRgn` and `FillRgn` procedures. You can erase it by using the `EraseRgn` procedure, and you can use the `InvertRgn` procedure to reverse the colors of the pixels within it. In all of these procedures, you refer to a region by the handle returned by the `NewRgn` function when you first created the region.

These routines depend on the local coordinate system of the current graphics port. If you draw a region in a graphics port different from the one in which you defined the region, it may not appear in the proper position in the graphics port.

▲ **WARNING**
If any horizontal or vertical line drawn through the region would intersect the region's outline more than 50 times, the results of these graphics operations are undefined. The `FrameRgn` procedure in particular requires that there would be no more than 25 such intersections. ▲

FrameRgn

To draw an outline inside a region, use the `FrameRgn` procedure.

```
PROCEDURE FrameRgn (rgn: RgnHandle);
```

`rgn` A handle to the region to frame.

DESCRIPTION

Using the current graphics port's pen pattern, pattern mode, and pen size, the `FrameRgn` procedure draws an outline just inside the region whose handle you pass in the `rgn` parameter. The outline never goes outside the region boundary. The pen location does not change.

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

SPECIAL CONSIDERATIONS

The `FrameRgn` procedure calls the routines `CopyRgn`, `InsetRgn`, and `DiffRgn`, so `FrameRgn` may temporarily use heap space that's three times the size of the original region.

The `FrameRgn` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

PaintRgn

To paint a region with the graphics pen's pattern and pattern mode, use the `PaintRgn` procedure.

```
PROCEDURE PaintRgn (rgn: RgnHandle);
```

`rgn` A handle to the region to paint.

DESCRIPTION

Using the pen pattern and pattern mode for the current graphics port, the `PaintRgn` procedure draws the interior of the region whose handle you pass in the `rgn` parameter. The pen location does not change.

SPECIAL CONSIDERATIONS

The `PaintRgn` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

You can use the `FillRgn` procedure, described next, to draw the interior of a region with a pen pattern different from that for the current graphics port.

FillRgn

To fill a region with any available bit pattern, use the `FillRgn` procedure.

```
PROCEDURE FillRgn (rgn: RgnHandle; pat: Pattern);
```

`rgn`	A handle to the region to fill.
`pat`	The bit pattern to use for the fill. Figure 3-3 on page 3-7 illustrates the default fill patterns and the constants you can use to represent them.

DESCRIPTION

Using the `patCopy` pattern mode, the `FillRgn` procedure draws the interior of the region (whose handle you pass in the `rgn` parameter) with the pattern defined in the `Pattern` record that you specify in the `pat` parameter.

This procedure leaves the location of the graphics pen unchanged.

SPECIAL CONSIDERATIONS

The `FillRgn` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

Listing 3-8 on page 3-28 and Listing 3-9 on page 3-29 illustrate how to use this procedure.

You can use the `GetPattern` and `GetIndPattern` routines, described on page 3-126 and page 3-127, respectively, to get a pattern stored in a resource. The `patCopy` pattern mode is described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8. The `Pattern` record is described on page 3-40.

You can use the `PaintRgn` procedure, described in the previous section, to draw the interior of a region with the pen pattern for the current graphics port. To fill a region with a pixel pattern, use the `FillCRegion` procedure, which is described in the chapter "Color QuickDraw."

EraseRgn

To erase a region, use the `EraseRgn` procedure.

```
PROCEDURE EraseRgn (rgn: RgnHandle);
```

`rgn`	The region to erase.

DESCRIPTION

Using the `patCopy` pattern mode, the `EraseRgn` procedure draws the interior of the region whose handle you pass in the `rgn` parameter with the background pattern for the current graphics port.

This procedure leaves the location of the graphics pen unchanged.

SPECIAL CONSIDERATIONS

The `EraseRgn` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

The `patCopy` pattern mode is described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8.

InvertRgn

To invert the pixels enclosed by a region, use the `InvertRgn` procedure.

```
PROCEDURE InvertRgn (rgn: RgnHandle);
```

`rgn` A handle to the region whose pixels are to invert.

DESCRIPTION

The `InvertRgn` procedure inverts the pixels enclosed by the region whose handle you pass in the `rgn` parameter. Every white pixel becomes black and every black pixel becomes white.

This procedure leaves the location of the graphics pen unchanged.

SPECIAL CONSIDERATIONS

The `InvertRgn` procedure was designed for 1-bit images in basic graphics ports. This procedure operates on color pixels in color graphics ports, but the results are predictable only with 1-bit or direct pixels. For indexed pixels, Color QuickDraw performs the inversion on the pixel indexes, which means the results depend entirely on the contents of the CLUT (which is described in the chapter "Color QuickDraw"). The eight colors used in basic QuickDraw are stored in a color table represented by the global variable `QDColors`. To display those eight basic QuickDraw colors on an indexed device, Color QuickDraw uses the Color Manager to obtain indexes to the colors in the CLUT that best map to the colors in the `QDColors` color table. Because the index, not the color value, is inverted, the results are unpredictable.

Inversion works better for direct pixels. Inverting a pure green, for example, that has red, green, and blue component values of $0000, $FFFF, and $0000 results in magenta, which has component values of $FFFF, $0000, and $FFFF.

The `InvertRgn` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

Scaling and Mapping Points, Rectangles, Polygons, and Regions

QuickDraw provides procedures to help you map points, rectangles, regions, and polygons from one rectangle to another. You can scale rectangles, regions, and polygons into other rectangles.

To derive vertical and horizontal scaling factors from the proportions of two rectangles, use the `ScalePt` procedure. To map a point in one rectangle to an equivalent position in another rectangle, use the `MapPt` procedure. To map and scale a rectangle within one rectangle to another rectangle, use the `MapRect` procedure. To map and scale a region within one rectangle to another rectangle, use the `MapRgn` procedure. To map and scale a polygon within one rectangle to another rectangle, use the `MapPoly` procedure.

If the points or rectangles supplied to these routines are defined in a graphics port other than your current graphics port, you must convert them to the local coordinate system of your current graphics port. You can accomplish this by using the `SetPort` procedure to change to the graphics port containing the points or rectangles, using the `LocalGlobal` procedure to convert their locations to global coordinates, using `SetPort` to return to your starting graphics port, and then using the `GlobalToLocal` procedure to convert the locations of points or rectangles to the local coordinates of your current graphics port. These procedures are described in the chapter “Basic QuickDraw.”

ScalePt

To scale a height and width according to the proportions of two rectangles, use the `ScalePt` procedure.

```
PROCEDURE ScalePt (VAR pt: Point; srcRect,dstRect: Rect);
```

`pt` On input, an initial height and width (specified in the two fields of a `Point` record) to scale; upon completion, vertical and horizontal scaling factors derived by multiplying the height and width by ratios of the height and width of the rectangle in the `srcRect` parameter to the height and width of the rectangle in the `dstRect` parameter.

`srcRect` A rectangle. The ratio of this rectangle’s height to the height of the rectangle in the `dstRect` parameter provides the vertical scaling factor, and the ratio of this rectangle’s width to the width of the rectangle in the `dstRect` parameter provides the horizontal scaling factor.

`dstRect` A rectangle compared to the rectangle in the `srcRect` parameter to determine vertical and horizontal scaling factors.

DESCRIPTION

The `ScalePt` procedure produces horizontal and vertical scaling factors from the proportions of two rectangles. You can use `ScalePt`, for example, to scale the dimensions of the graphics pen.

You specify an initial height and width to scale in the `pt` parameter. This parameter is of type `Point`, although you don't pass coordinates in this parameter. Instead, you pass an initial height to scale in the `v` (or vertical) field of the `Point` record, and you pass an initial width to scale in the `h` (or horizontal) field.

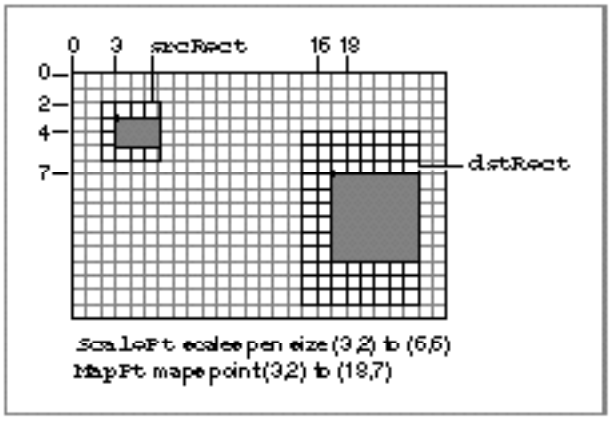
The `ScalePt` procedure scales these measurements by multiplying the initial height you specify in the `pt` parameter by the ratio of the height of the rectangle you specify in the `dstRect` parameter to the height of the rectangle you specify in the `srcRect` parameter, and by multiplying the initial width in the `pt` parameter by the ratio of the width of the `dstRect` rectangle to the width of the `srcRect` rectangle. The `ScalePt` procedure returns the result in the `pt` parameter.

In Figure 3-23, where the width of the `dstRect` rectangle is twice the width of the `srcRect` rectangle, and its height is three times the height of `srcRect`, `ScalePt` scales the width of the graphics pen from 3 to 6 and scales its height from 2 to 6.

SPECIAL CONSIDERATIONS

The minimum value `ScalePt` returns is (1,1).

Figure 3-23 Using `ScalePt` and `MapPt`



MapPt

To map a point in one rectangle to an equivalent position in another rectangle, use the `MapPt` procedure.

```
PROCEDURE MapPt (VAR pt: Point; srcRect,dstRect: Rect);
```

`pt`	Upon input, the point in the source rectangle to map; upon completion, its mapped position in the destination rectangle.
`srcRect`	The source rectangle containing the original point.
`dstRect`	The destination rectangle in which the point will be mapped.

DESCRIPTION

The `MapPt` procedure maps a point in one rectangle to an equivalent position in another rectangle.

In the `pt` parameter, you specify a point that lies within the rectangle that you specify in the `srcRect` parameter. The `MapPt` procedure maps this point to a similarly located point within the rectangle that you specify in the `dstRect` parameter—that is, to where it would fall if it were part of a drawing being expanded or shrunk to fit the destination rectangle. The `MapPt` procedure returns the location of the mapped point in the `pt` parameter. For example, a corner point of the source rectangle would be mapped to the corresponding corner point of the destination rectangle in `dstRect`, and the center of the source rectangle would be mapped to the center of destination rectangle.

The source and destination rectangles may overlap, and the point you specify need not actually lie within the source rectangle.

In Figure 3-23 on page 3-105, the point (3,2) in the source rectangle is mapped to (18,7) in the destination rectangle.

SEE ALSO

If you're going to draw inside the destination rectangle, you'll probably also want to scale the graphics pen size accordingly with the `ScalePt` procedure, described in the previous section.

MapRect

To map and scale a rectangle within one rectangle to another rectangle, use the `MapRect` procedure.

```
PROCEDURE MapRect (VAR r: Rect; srcRect,dstRect: Rect);
```

r	Upon input, the rectangle to map; upon completion, the mapped rectangle.
srcRect	The rectangle containing the rectangle to map.
dstRect	The rectangle in which the new rectangle will be mapped.

DESCRIPTION

The MapRect procedure takes a rectangle within one rectangle and maps and scales it to another rectangle. In the r parameter, you specify a rectangle that lies within the rectangle that you specify in the srcRect parameter. By calling the MapPt procedure to map the upper-left and lower-right corners of the rectangle in the r parameter, MapRect maps and scales it to the rectangle that you specify in the dstRect parameter. The MapRect procedure returns the newly mapped rectangle in the r parameter.

MapRgn

To map and scale a region within one rectangle to another rectangle, use the MapRgn procedure.

```
PROCEDURE MapRgn (rgn: RgnHandle; srcRect,dstRect: Rect);
```

rgn	A handle to a region. Upon input, this is the region to map. Upon completion, this region is the one mapped to a new location.
srcRect	The rectangle containing the region to map.
dstRect	The rectangle in which the new region will be mapped.

DESCRIPTION

The MapRgn procedure takes a region within one rectangle and maps and scales it to another rectangle. In the rgn parameter, you specify a handle to a region that lies within the rectangle that you specify in the srcRect parameter. By calling the MapPt procedure to map all the points of the region in the rgn parameter, MapRgn maps and scales it to the rectangle that you specify in the dstRect parameter. The MapRgn procedure returns the result in the region whose handle you initially passed in the rgn parameter.

The MapRgn procedure is useful for determining whether a region operation will exceed available memory. By mapping a large region into a smaller one and performing the operation (without actually drawing), you can estimate how much memory will be required by the anticipated operation.

SPECIAL CONSIDERATIONS

The MapRgn procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

MapPoly

To map and scale a polygon within one rectangle to another rectangle, use the `MapPoly` procedure.

```
PROCEDURE MapPoly (poly: PolyHandle; srcRect,dstRect: Rect);
```

`poly`	A handle to a polygon. Upon input, this is the polygon to map. Upon completion, this polygon is the one mapped to a new location.
`srcRect`	The rectangle containing the polygon.
`dstRect`	The rectangle in which the new region will be mapped.

DESCRIPTION

The `MapPoly` procedure takes a polygon within one rectangle and maps and scales it to another rectangle. In the `poly` parameter, you specify a handle to a polygon that lies within the rectangle that you specify in the `srcRect` parameter. By calling the `MapPt` procedure to map all the points that define the polygon specified in the `poly` parameter, `MapPoly` maps and scales it to the rectangle that you specify in the `dstRect` parameter. The `MapPoly` procedure returns the result in the polygon whose handle you initially passed in the `poly` parameter.

Similar to the `MapRgn` procedure described in the previous section, the `MapPoly` procedure is useful for determining whether a polygon operation will exceed available memory.

Calculating Black-and-White Fills

QuickDraw provides the `SeedFill` and `CalcMask` procedures to help you determine the results of filling operations on portions of bitmaps. (Procedures for determining filling operations on pixel maps—namely, `SeedCFill` and `CalcCMask`—are described in the chapter "Color QuickDraw.")

The `SeedFill` procedure produces a mask showing where bits would be filled from a starting point, like the paint pouring from the MacPaint® paint-bucket tool. The `CalcMask` procedure produces a mask showing where paint could *not* flow from any of the outer edges of a rectangle. You can use the resulting masks to transfer portions of bit images from one graphics port to another with the `CopyBits` or `CopyMask` procedure, both of which are described in "Copying Images" beginning on page 3-112.

SeedFill

To determine how far filling will extend from a seeding point, use the `SeedFill` procedure.

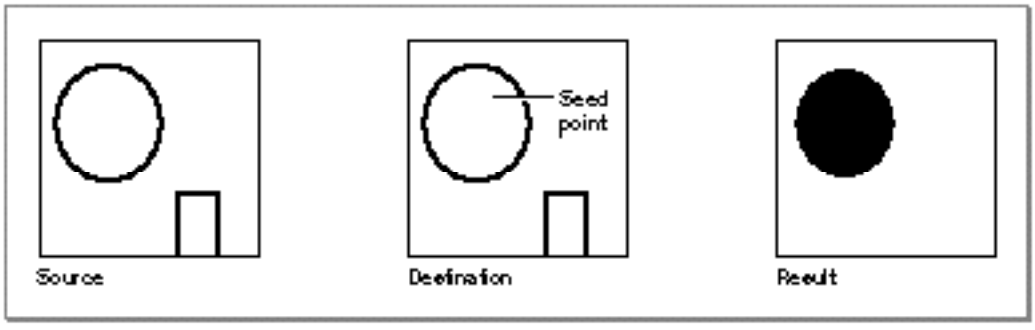
```
PROCEDURE SeedFill (srcPtr,dstPtr: Ptr;
                    srcRow,dstRow,height,words,
                    seedH,seedV: Integer);
```

`srcPtr`	A pointer to the source bit image.
`dstPtr`	On input, a pointer to the destination bit image; upon return, a pointer to the bitmap containing the resulting mask.
`srcRow`	Row width of the source bitmap.
`dstRow`	Row width of the destination bitmap.
`height`	Height (in pixels) of the fill rectangle.
`words`	Width (in words) of the fill rectangle.
`seedH`	The horizontal offset (in pixels) at which to begin filling the destination bit image.
`seedV`	The vertical offset (in pixels) at which to begin filling the destination bit image.

DESCRIPTION

The `SeedFill` procedure produces a mask showing where bits in an image can be filled from a starting point, like the paint pouring from the MacPaint paint-bucket tool. The `SeedFill` returns this mask in the `dstPtr` parameter. This mask is a bitmap filled with 1's only where the pixels in the source image can be filled. This is illustrated in Figure 3-24. You can then use this mask with the `CopyBits`, `CopyMask`, and `CopyDeepMask` procedures.

Figure 3-24 A source image and its resulting mask produced by the `SeedFill` procedure

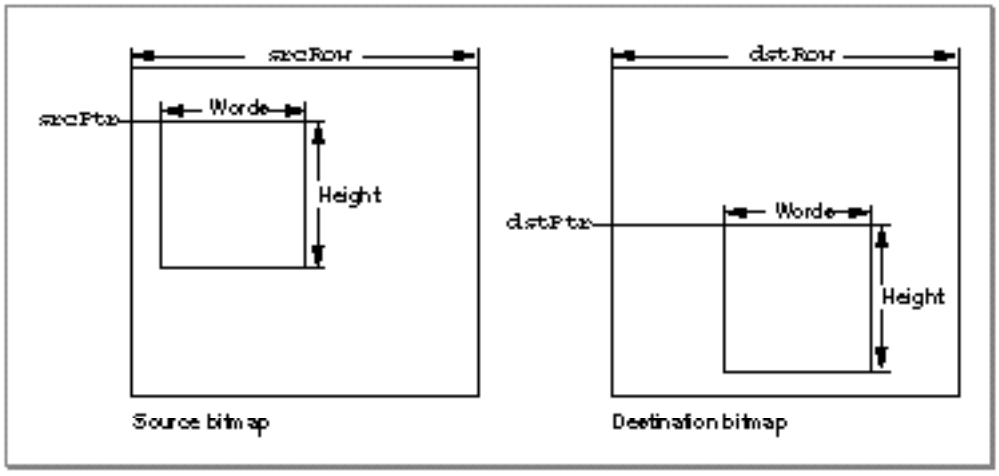


Point to the bit image you want to fill with the `srcPtr` parameter, which can point to the image's base address or a word boundary within the image. Specify a pixel height and word width with the `height` and `words` parameters to define a fill rectangle that delimits the area you want to fill. The fill rectangle can be the entire bit image or a subset of it. Point to a destination image with the `dstPtr` parameter. Specify the row widths of the source and destination bitmaps (their `rowBytes` values) with the `srcRow` and `dstRow` parameters. (The bitmaps can be different sizes, but they must be large enough to contain the fill rectangle at the origins specified by the `srcPtr` and `dstPtr` parameters.) Figure 3-25 illustrates these parameters for the source and destination bit images.

You specify where to begin filling with the `seedH` and `seedV` parameters: they specify a horizontal and vertical offset in pixels from the origin of the image pointed to by the `srcPtr` parameter. The `SeedFill` procedure calculates contiguous pixels from that point out to the boundaries of the fill rectangle, and it stores the result in the bit image pointed to by the `dstPtr` parameter.

Calls to `SeedFill` are not clipped to the current port and are not stored into QuickDraw pictures.

Figure 3-25 Parameters for the `SeedFill` and `CalcMask` procedures



SEE ALSO

For color graphics ports, use the `SeedCFill` procedure, which is described in the chapter "Color QuickDraw."

CalcMask

To determine where filling will not occur when filling from the outside of a rectangle, use the `CalcMask` procedure.

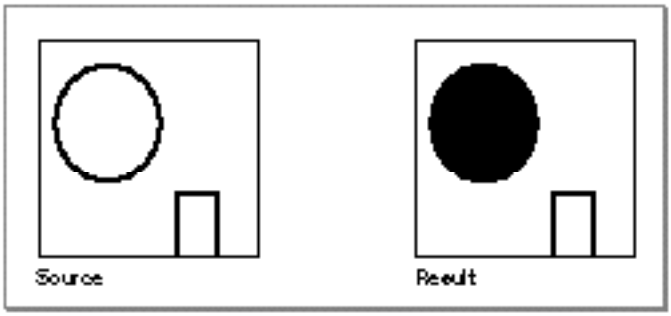
```
PROCEDURE CalcMask (srcPtr,dstPtr: Ptr;
                    srcRow,dstRow,height,words: Integer);
```

`srcPtr`	A pointer to the source bit image.
`dstPtr`	A pointer to the destination bit image.
`srcRow`	Row width of the source bitmap.
`dstRow`	Row width of the destination bitmap.
`height`	Height (in pixels) of the fill rectangle.
`words`	Width (in words) of the fill rectangle.

DESCRIPTION

The `CalcMask` procedure produces a bit image with 1's in all pixels to which paint could *not* flow from any of the outer edges of the rectangle. You can use this bit image as a mask with the `CopyBits` or `CopyMask` procedure. As illustrated in Figure 3-26, a hollow object produces a solid mask, but an open object produces a mask of itself.

Figure 3-26 A source image and the resulting mask produced by the `CalcMask` procedure



As with the `SeedFill` procedure, point to the bit image you want to fill with the `srcPtr` parameter, which can point to the image's base address or a word boundary within the image. Specify a pixel height and word width with the `height` and `words` parameters to define a fill rectangle that delimits the area you want to fill. The fill rectangle can be the entire bit image or a subset of it. Point to a destination image with the `dstPtr` parameter. Specify the row widths of the source and destination bitmaps (their `rowBytes` values) with the `srcRow` and `dstRow` parameters. (The bitmaps can be different sizes, but they must be large enough to contain the fill rectangle at the origins specified by `srcPtr` and `dstPtr`.)

Figure 3-25 on page 3-110 illustrates the parameters for the source and destination bit images.

Calls to `CalcMask` are not clipped to the current port and are not stored into QuickDraw pictures.

SEE ALSO

For color graphics ports, use the `CalcCMask` procedure, which is described in the chapter "Color QuickDraw."

Copying Images

QuickDraw provides three procedures for copying portions of bitmaps from one graphics port or offscreen graphics world into another graphics port. The `CopyBits` procedure allows you to copy using source modes and to clip and resize during the copy operation. The `CopyMask` procedure allows you to mask areas where you want the copy operation to occur. These procedures also allow you to use pixel maps instead of bitmaps when your application runs on systems supporting Color QuickDraw.

The `CopyDeepMask` procedure, which is available to basic QuickDraw only in System 7, combines the functionality of both `CopyBits` and `CopyMask`.

CopyBits

You can use the `CopyBits` procedure to copy a portion of a bitmap or a pixel map from one graphics port (or offscreen graphics world) into another graphics port.

```
PROCEDURE CopyBits (srcBits,dstBits: BitMap;
                    srcRect,dstRect: Rect; mode: Integer;
                    maskRgn: RgnHandle);
```

`srcBits`	The source `BitMap` record.
`dstBits`	The destination `BitMap` record.
`srcRect`	The source rectangle.
`dstRect`	The destination rectangle.
`mode`	One of the eight source modes in which the copy is to be performed.
`maskRgn`	A region to use as a clipping mask.

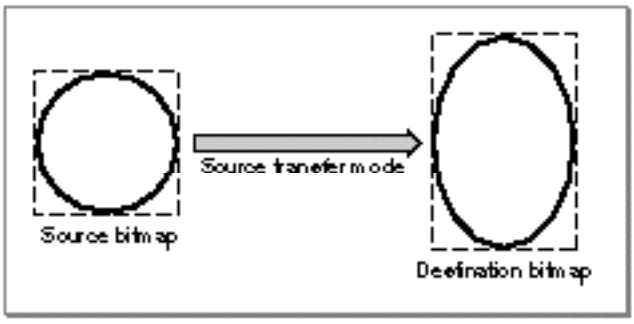
DESCRIPTION

The `CopyBits` procedure transfers any portion of a bitmap between two basic graphics ports, or any portion of a pixel map between two color graphics ports. You can use `CopyBits` to move offscreen graphic images into an onscreen window, to blend colors for the image in a pixel map, and to shrink and expand images.

Specify a source bitmap in the `srcBits` parameter and a destination bitmap in the `dstBits` parameter. When copying images between color graphics ports, you must coerce each `CGrafPort` record to a `GrafPort` record, dereference the `portBits` fields of each, and then pass these "bitmaps" in the `srcBits` and `dstBits` parameters. If your application copies a pixel image from a color graphics port called `MyColorPort`, for example, you could specify `GrafPtr(MyColorPort)^.portBits` in the `srcBits` parameter. In a `CGrafPort` record, the high 2 bits of the `portVersion` field are set. This field, which shares the same position in a `CGrafPort` record as the `portBits.rowBytes` field in a `GrafPort` record, indicates to `CopyBits` that you have passed it a handle to a pixel map rather than a bitmap.

Using the `srcRect` and `dstRect` parameters, you can specify identically or differently sized source and destination rectangles; for differently sized rectangles, `CopyBits` scales the source image to fit the destination. As shown in Figure 3-27, for example, if the bit image is a circle in a square source rectangle, and the destination rectangle is not square, the bit image appears as an oval in the destination. When you specify rectangles in the `srcRect` and `dstRect` parameters, use the local coordinate systems of, respectively, the source and destination graphics ports.

Figure 3-27 Using `CopyBits` to stretch an image



In the mode parameter, specify one of the following source modes for transferring the bits from a source bitmap to a destination bitmap:

```
CONST              {source modes for basic graphics ports}
   srcCopy   = 0; {where source pixel is black, force }
                  { destination pixel black; where source pixel }
                  { is white, force destination pixel white}
   srcOr     = 1; {where source pixel is black, force }
                  { destination pixel black; where source pixel }
                  { is white, leave destination pixel unaltered}
   srcXor    = 2; {where source pixel is black, invert }
                  { destination pixel; where source pixel is }
                  { white, leave destination pixel unaltered}
   srcBic    = 3; {where source pixel is black, force }
                  { destination pixel white; where source pixel }
                  { is white, leave destination pixel unaltered}
   notSrcCopy
             = 4; {where source pixel is black, force }
                  { destination pixel white; where source pixel }
                  { is white, force destination pixel black}
   notSrcOr  = 5; {where source pixel is black, leave }
                  { destination pixel unaltered; where source }
                  { pixel is white, force destination pixel black}
   notSrcXor = 6; {where source pixel is black, leave }
                  { destination pixel unaltered; where source }
                  { pixel is white, invert destination pixel}
   notSrcBic = 7; {where source pixel is black, leave }
                  { destination pixel unaltered; where source }
                  { pixel is white, force destination pixel white}
```

On computers running System 7, you can add dithering to any source mode by adding the following constant or the value it represents to the source mode:

```
CONST ditherCopy  = 64; {add to source mode for dithering}
```

Dithering is a technique that mixes existing colors to create the effect of additional colors. It also improves images that you shrink or that you copy from a direct pixel device to an indexed device. The CopyBits procedure always dithers images when shrinking them between pixel maps on direct devices.

To use highlighting, you can add this constant or its value to the source mode:

```
CONST hilite= 50; {add to source or pattern mode for highlighting}
```

With highlighting, QuickDraw replaces the background color with the highlight color when your application copies images between graphics ports. This has the visual effect of using a highlighting pen to select the object. (The global variable `HiliteRGB` is read from parameter RAM when the machine starts. Basic graphics ports use the color stored in the `HiliteRGB` global variable as the highlight color. Color graphics ports default to the `HiliteRGB` global variable, but can be overridden by the `HiliteColor` procedure, described in the chapter "Color QuickDraw.")

When transferring pixels from a source pixel map to a destination pixel map, Color QuickDraw interprets the source mode constants differently than basic QuickDraw does. These constants have the following effects under Color QuickDraw:

```
CONST               {source modes for color graphics ports}
   srcCopy    = 0;{determine how close the color of the source }
                  { pixel is to black, and assign this relative }
                  { amount of foreground color to the }
                  { destination pixel; determine how close the }
                  { color of the source pixel is to white, and }
                  { assign this relative amount of background }
                  { color to the destination pixel}
   srcOr      = 1;{determine how close the color of the source }
                  { pixel is to black, and assign this relative }
                  { amount of foreground color to the }
                  { destination pixel}
   srcXor     = 2;{where source pixel is black, invert the }
                  { destination pixel--for a colored destination }
                  { pixel, use the complement of its color }
                  { if the pixel is direct, invert its index if }
                  { the pixel is indexed}
   srcBic     = 3;{determine how close the color of the source }
                  { pixel is to black, and assign this relative }
                  { amount of background color to the }
                  { destination pixel}
   notSrcCopy = 4;{determine how close the color of the source }
                  { pixel is to black, and assign this relative }
                  { amount of background color to the }
                  { destination pixel; determine how close the }
                  { color of the source pixel is to white, and }
                  { assign this relative amount of foreground }
                  { color to the destination pixel}
   notSrcOr   = 5; {determine how close the color of the source }
                  { pixel is to white, and assign this relative }
                  { amount of foreground color to the }
                  { destination pixel}
```

```
   notSrcXor = 6; {where source pixel is white, invert the }
                  { destination pixel--for a colored destination }
                  { pixel, use the complement of its color }
                  { if the pixel is direct, invert its index if }
                  { the pixel is indexed}
   notSrcBic = 7; {determine how close the color of the source }
                  { pixel is to white, and assign this relative }
                  { amount of background color to the }
                  { destination pixel}
```

When you use `CopyBits` on a computer running Color QuickDraw, you can also specify one of the following transfer modes in the `mode` parameter:

```
CONST {arithmetic transfer modes available in Color QuickDraw}
   blend       = 32; {replace destination pixel with a blend }
                     { of the source and destination pixel }
                     { colors; if the destination is a bitmap or }
                     { 1-bit pixel map, revert to srcCopy mode}
   addPin      = 33; {replace destination pixel with the sum of }
                     { the source and destination pixel colors-- }
                     { up to a maximum allowable value; if }
                     { the destination is a bitmap or }
                     { 1-bit pixel map, revert to srcBic mode}
   addOver     = 34; {replace destination pixel with the sum of }
                     { the source and destination pixel colors-- }
                     { but if the value of the red, green, or }
                     { blue component exceeds 65,536, then }
                     { subtract 65,536 from that value; if the }
                     { destination is a bitmap or 1-bit }
                     { pixel map, revert to srcXor mode}
   subPin      = 35; {replace destination pixel with the }
                     { difference of the source and destination }
                     { pixel colors--but not less than a minimum }
                     { allowable value; if the destination }
                     { is a bitmap or 1-bit pixel map, revert to }
                     { srcOr mode}
   transparent = 36; {replace the destination pixel with the }
                     { source pixel if the source pixel isn't }
                     { equal to the background color}
```

```
addMax      = 37; {compare the source and destination pixels, }
                  { and replace the destination pixel with }
                  { the color containing the greater }
                  { saturation of each of the RGB components; }
                  { if the destination is a bitmap or }
                  { 1-bit pixel map, revert to srcBic mode}
subOver     = 38; {replace destination pixel with the }
                  { difference of the source and destination }
                  { pixel colors--but if the value of the }
                  { red, green, or blue component is }
                  { less than 0, add the negative result to }
                  { 65,536; if the destination is a bitmap or }
                  { 1-bit pixel map, revert to srcXor mode}
adMin       = 39; {compare the source and destination pixels, }
                  { and replace the destination pixel with }
                  { the color containing the lesser }
                  { saturation of each of the RGB components; }
                  { if the destination is a bitmap or }
                  { 1-bit pixel map, revert to srcOr mode}
```

You can pass a region handle in the `MaskRgn` parameter to specify a mask region; the resulting image is always clipped to this mask region and to the boundary rectangle of the destination bitmap. If the destination bitmap is the current graphics port's bitmap, it's also clipped to the intersection of the graphics port's clipping region and visible region. If you don't want to clip to a masking region, just pass `NIL` for the `maskRgn` parameter.

SPECIAL CONSIDERATIONS

When you use the `CopyBits` procedure to transfer an image between pixel maps, the source and destination images may be of different pixel depths, of different sizes, and they may have different color tables. However, `CopyBits` assumes that the destination pixel map uses the same color table as the color table for the current `GDevice` record. (This is because the Color Manager requires an inverse table for translating the color table from the source pixel map to the destination pixel map.)

The `CopyBits` procedure applies the foreground and background colors of the current graphics port to the image in the destination pixel map (or bitmap), even if the source image is a bitmap. This causes the foreground color to replace all black pixels in the destination and the background color to replace all white pixels. To avoid unwanted coloring of the image, use the `RGBForeColor` procedure to set the foreground to black and use the `RGBBackColor` procedure to set the background to white before calling `CopyBits`.

The source bitmap or pixel map must not occupy more memory than half the available stack space. The stack space required by `CopyBits` is roughly five times the value of the `rowBytes` field of the source pixel map: one `rowBytes` value for the pixel map (or bitmap), an additional `rowBytes` value for dithering, another `rowBytes` value when stretching or shrinking the source pixel map into the destination, another `rowBytes` value for any color map changing, and a fifth additional `rowBytes` value for any color aliasing. If there is insufficient memory to complete a `CopyBits` operation in Color QuickDraw, the `QDError` function (described in the chapter "Color QuickDraw" in this book) returns the result code –143.

If you use `CopyBits` to copy between two graphics ports that overlap, you must first use the `LocalToGlobal` procedure to convert to global coordinates, and then specify the global variable `screenBits` for both the `srcBits` and `dstBits` parameters.

The `CopyBits` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

If you are reading directly from a NuBus™ video card with a base address of Fs00000 and there is not a card in the slot (s–1) below it, `CopyBits` reads addresses less than the base address of the pixel map. This causes a bus error. To work around the problem, remap the `baseAddr` field of the pixel map in your video card to at least 20 bytes above the NuBus boundary; an address link of Fs000020 precludes the problem.

SEE ALSO

Listing 3-11 on page 3-33 illustrates how to use `CopyBits` to scale an image when copying it from one window into another. Source modes are described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8. Plate 2 at the front of this book illustrates how to use `CopyBits` to colorize an image in a color graphics port; Listing 4-5 on page 4-35 in the chapter "Color QuickDraw" shows the sample code that produced this plate. Listing 6-1 on page 6-5 in the chapter "Offscreen Graphics Worlds" illustrates how to use `CopyBits` to copy an image from an offscreen graphics world to an onscreen color graphics port.

Dithering, pixel maps, color graphics ports, the `RGBForeColor` and `RGBBackColor` procedures, and color tables are explained in the chapter "Color QuickDraw." The `LocalToGlobal` procedure is described in the chapter "Basic QuickDraw." The `GDevice` record is described in the chapter "Graphics Devices." Inverse tables and the Color Manager are described in the chapter "Color Manager" in *Inside Macintosh: Advanced Color Imaging.*

"Copying Pixels Between Color Graphics Ports" in the chapter "Color QuickDraw" describes in greater detail how to use `CopyBits` to transfer colored images.

The `CopyDeepMask` procedure (described on page 3-120) combines the functions of the `CopyBits` and `CopyMask` procedures. (The `CopyMask` procedure is described next.)

CopyMask

You can use the `CopyMask` procedure to copy a bit or pixel image from one graphics port (or offscreen graphics world) into another graphics port only where the bits in a mask are set to 1.

```
PROCEDURE CopyMask (srcBits,maskBits,dstBits: BitMap;
                    srcRect,maskRect,dstRect: Rect);
```

`srcBits`	The source `BitMap` record.
`maskBits`	The mask `BitMap` record.
`dstBits`	The destination `BitMap` record.
`srcRect`	The source rectangle.
`maskRect`	The mask rectangle. This must be the same size as the rectangle passed in the `srcRect` parameter.
`dstRect`	The destination rectangle.

DESCRIPTION

The `CopyMask` procedure copies the source bitmap or pixel map that you specify in the `srcBits` parameter to a destination bitmap or pixel map that you specify in the `dstBits` parameter—but only where the bits of the mask bitmap or pixel map that you specify in the `maskBits` parameter are set to 1. When copying images between color graphics ports, you must coerce each `CGrafPort` record to a `GrafPort` record, dereference the `portBits` fields of each, and then pass these "bitmaps" in the `srcBits` and `dstBits` parameters. If your application copies a pixel image from a color graphics port called `MyColorPort`, for example, you could specify `GrafPtr(MyColorPort)^.portBits` in the `srcBits` parameter.

Using the `srcRect` and `dstRect` parameters, you can specify identically or differently sized source and destination rectangles; for differently sized rectangles, `CopyMask` scales the source image to fit the destination. When you specify rectangles in the `srcRect` and `dstRect` parameters, use the local coordinate systems of, respectively, the source and destination graphics ports.

The rectangle you pass in the `maskRect` parameter selects the portion of the bitmap or pixel map that you specify in the `maskBits` parameter to use as the mask.

If you specify pixel maps to `CopyMask`, they may range from 1 to 32 pixels in depth. The pixel depth of the mask that you specify in the `maskBits` parameter is applied as a filter between the source and destination pixel maps that you specify in the `srcBits` and `dstBits` parameters. A black mask pixel value means that the copy operation is to take the source pixel; a white value means that the copy operation is to take the destination

pixel. Intermediate values specify a weighted average, which is calculated on a color component basis. For each pixel's color component value, the calculation is

(1 – mask) × source + (mask) × destination

Thus high mask values for a pixel's color component reduce that component's contribution from the source `PixMap` record.

SPECIAL CONSIDERATIONS

Calls to `CopyMask` are not recorded in pictures and do not print.

See the list of special considerations for the `CopyBits` procedure beginning on page 3-117; these considerations also apply to `CopyMask`.

The `CopyMask` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

You can use the bitmap returned by the `CalcMask` procedure, described on page 3-111, as the mask in order to implement a mask copy similar to that performed by the MacPaint lasso tool. In the same way, you could use the pixel map returned by the `CalcCMask` procedure, described in the chapter "Color QuickDraw."

The chapter "Color QuickDraw" describes in more detail how to use `CopyMask` in a Color QuickDraw environment. Plate 3 at the front of this book illustrates how to use different colors in the mask to produce different effects in the destination pixel map; Listing 6-2 on page 6-10 in the chapter "Offscreen Graphics Worlds" shows the code that produced this plate. Plate 4 at the front of this book provides another illustration of the effects of the source and mask pixel maps on the destination pixel map.

The `CopyDeepMask` procedure (described next) combines the functions of the `CopyMask` and `CopyBits` procedures.

CopyDeepMask

To use a mask when copying bitmaps or pixel maps between graphics ports (or from an offscreen graphics world into a graphics port), you can use the `CopyDeepMask` procedure, which combines the effects of the `CopyBits` and `CopyMask` procedures.

```
PROCEDURE CopyDeepMask  (srcBits: BitMap; maskBits: BitMap;
                         dstBits: BitMap; srcRect: Rect;
                         maskRect: Rect; dstRect: Rect;
                         mode: Integer; maskRgn: RgnHandle);
```

srcBits	The source BitMap record.
maskBits	The masking BitMap record.
dstBits	The destination BitMap record.
srcRect	The source rectangle.
maskRect	The mask rectangle. This must be the same size as the rectangle passed in the srcRect parameter.
dstRect	The destination rectangle.
mode	The source mode.
maskRgn	The mask clipping region.

DESCRIPTION

The CopyDeepMask procedure transfers a bitmap between two basic graphics ports or a pixel map between two color graphics ports. You specify a mask to CopyDeepMask so that it transfers the source image to the destination image only where the bits of the mask are set to 1. You can use CopyDeepMask to move offscreen graphic images into an onscreen window, to blend colors for the image in a pixel map, and to shrink and expand images.

Specify a source bitmap in the srcBits parameter and a destination bitmap in the dstBits parameter. Specify a mask in the maskBits parameter. When copying images between color graphics ports, you must coerce each port's CGrafPort record to a GrafPort record, dereference the portBits fields of each, and then pass these "bitmaps" in the srcBits and dstBits parameters. If your application copies a pixel image from a color graphics port called MyColorPort, for example, you could specify GrafPtr(MyColorPort)^.portBits in the srcBits parameter. The transfer can be performed in any of the transfer modes—with or without adding the ditherCopy constant—that are available to the CopyBits procedure, described beginning on page 3-112.

Using the srcRect and dstRect parameters, you can specify identically or differently sized source and destination rectangles; for differently sized rectangles, CopyDeepMask scales the source image to fit the destination. When you specify rectangles in the srcRect and dstRect parameters, use the local coordinate systems of, respectively, the source and destination graphics ports.

The result (in the parameter dstBits) is clipped to the mask region that you specify in the maskRgn parameter, and to the boundary rectangle that you specify in the dstRect parameter. The rectangle you pass in the maskRect parameter selects the portion of the bitmap or pixel map that you specify in the maskBits parameter to use as the mask. If you don't want to clip to the mask region, specify NIL in the maskRgn parameter.

If you specify pixel maps to `CopyDeepMask`, they may range from 1 to 32 pixels in depth. The pixel depth of the mask that you specify in the `maskBits` parameter is applied as a filter between the source and destination pixel maps that you specify in the `srcBits` and `dstBits` parameters. A black mask pixel value means that the copy operation is to take the source pixel; a white value means that the copy operation is to take the destination pixel. Intermediate values specify a weighted average, which is calculated on a color component basis. For each pixel's color component value, the calculation is

(1 – mask) × source + (mask) × destination

Thus high mask values for a pixel's color component reduce that component's contribution from the source `PixMap` record.

SPECIAL CONSIDERATIONS

This procedure is available to basic QuickDraw only in System 7.

As with the `CopyMask` procedure, calls to `CopyDeepMask` are not recorded in pictures and do not print.

See the list of special considerations for the `CopyBits` procedure beginning on page 3-117; these considerations also apply to `CopyDeepMask`.

The `CopyDeepMask` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

The chapter "Color QuickDraw" describes in more detail how to use `CopyDeepMask` in a Color QuickDraw environment.

Drawing With the Eight-Color System

On a color screen, you can draw using eight predefined colors, even when you are using a basic graphics port. Although basic QuickDraw graphics routines were designed for black-and-white drawing, they also include rudimentary color capabilities. Because Color QuickDraw also supports this system, it is compatible across all Macintosh platforms. (This section describes the rudimentary color routines included in basic QuickDraw. See the next chapter, "Color QuickDraw," for information about more sophisticated color use in your application.)

A pair of fields in a `GrafPort` record, `fgColor` and `bkColor`, specify a foreground and background color. The foreground color is the color of the "ink" used to frame, fill, and paint. By default, the foreground color is black. The background color is the color of the pixels in the bitmap wherever no drawing has taken place. By default, the background color is white. However, you can use the `ForeColor` and `BackColor` procedures to change these fields. When printing, however, use the `ColorBit` procedure to set the foreground color.

In System 7, these Color QuickDraw routines are available to basic QuickDraw: `RGBForeColor`, `RGBBackColor`, `GetForeColor`, and `GetBackColor`. Described in the next chapter, "Color QuickDraw," these routines can also assist you in manipulating the eight-color system of basic QuickDraw.

ForeColor

To change the color of the "ink" used for framing, painting, and filling on computers that support only basic QuickDraw, you can use the `ForeColor` procedure.

```
PROCEDURE ForeColor (color: LongInt);
```

`color` One of eight color values. You can use the following constants to represent these values:

```
CONST
   whiteColor     = 30;
   blackColor     = 33;
   yellowColor    = 69;
   magentaColor   = 137;
   redColor       = 205;
   cyanColor      = 273;
   greenColor     = 341;
   blueColor      = 409;
```

DESCRIPTION

The `ForeColor` procedure sets the foreground color for the current graphics port to the color that you specify in the `color` parameter. When you draw with the `patCopy` and `srcCopy` transfer modes, for example, black pixels are drawn in the color you specify with `ForeColor`.

SPECIAL CONSIDERATIONS

The `ForeColor` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

All nonwhite colors appear as black on black-and-white screens. Before you use `ForeColor`, you can use the `DeviceLoop` procedure, which is described in the chapter "Graphics Devices," to determine the color characteristics of the current screen.

In System 7, you may instead use the Color QuickDraw procedure `RGBForeColor`, which is described in the chapter "Color QuickDraw."

BackColor

To change a basic graphics port's background color, use the `BackColor` procedure.

```
PROCEDURE BackColor (color: LongInt);
```

`color` One of eight color values. You can use the constants described for the `ForeColor` procedure in the previous section.

DESCRIPTION

The `BackColor` procedure sets the background color for the current graphics port to the color that you specify in the `color` parameter. When you draw with the `patCopy` and `srcCopy` transfer modes, for example, white pixels are drawn in the color you specify with `BackColor`.

SPECIAL CONSIDERATIONS

The `BackColor` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

All nonwhite colors appear as black on black-and-white screens. Before you use `BackColor`, you can use the `DeviceLoop` procedure, which is described in the chapter "Graphics Devices," to determine the color characteristics of the current screen.

In System 7, you may instead use the Color QuickDraw procedure `RGBBackColor`, which is described in the chapter "Color QuickDraw."

ColorBit

Use the `ColorBit` procedure to set the foreground color for all printing in the current graphics port.

```
PROCEDURE ColorBit (whichBit: Integer);
```

`whichBit` An integer specifying the plane to draw into.

DESCRIPTION

The `ColorBit` procedure is called by printing software for a color printer (or other color-imaging software) to set the `GrafPort` record's `colorBit` field to the value in the `whichBit` parameter. This value tells QuickDraw which plane of the color picture to draw into. QuickDraw draws into the plane corresponding to the bit number specified by the `whichBit` parameter. Since QuickDraw can support output devices that have up to 32 bits of color information per pixel, the possible range of values for `whichBit` is 0 through 31. The initial value of the `colorBit` field is 0.

Determining Whether QuickDraw Has Finished Drawing

Your application can use the `QDDone` function to determine whether drawing is completed in a given graphics port. You can also use it to determine whether drawing has finished in all open graphics ports.

QDDone

Although you will probably never need to determine whether QuickDraw has completed drawing, you can do so by using the `QDDone` function.

```
FUNCTION QDDone (port: GrafPtr): Boolean;
```

`port` The `GrafPort` record for a graphics port in which your application has begun drawing; if you pass `NIL`, `QDDone` tests all open graphics ports.

DESCRIPTION

The `QDDone` function returns `TRUE` if all drawing operations have finished in the graphics port specified in the `port` parameter, `FALSE` if any remain to be executed. If you pass `NIL` in the `port` parameter, then `QDDone` returns `TRUE` only if drawing operations have completed in all ports.

The `QDDone` function may be useful if a graphics accelerator is present and operating asynchronously. You could use it to ensure that all drawing is done before issuing new drawing commands, and to avoid the possibility that the new drawing operations might be overlaid by previously issued but unexecuted operations.

SPECIAL CONSIDERATIONS

The `QDDone` function has little or no usefulness.

If a graphics port draws a clock or some other continuously operating drawing process, `QDDone` may never return `TRUE`.

To determine whether all drawing in a color graphics port has completed, you must coerce its `CGrafPort` record to a `GrafPort` record, which you pass in the `port` parameter.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `QDDone` function are

Trap macro	Selector
`_QDExtensions`	$00040013

Getting Pattern Resources

As described in "Bit Patterns" beginning on page 3-5, QuickDraw predefines five patterns for your use in the global variables `white`, `black`, `gray`, `ltGray`, and `dkGray`. However, you can create and store your own patterns in a resource file. To retrieve the patterns stored in a pattern (`'PAT '`) resource, you can use the `GetPattern` function. To retrieve the patterns stored in a pattern list (`'PAT#'`) resource, you can use the `GetIndPattern` procedure.

GetPattern

To get a pattern (`'PAT '`) resource stored in a resource file, you can use the `GetPattern` function.

```
FUNCTION GetPattern (patID: Integer): PatHandle;
```

`patID` The resource ID for a resource of type `'PAT '`.

DESCRIPTION

The `GetPattern` function returns a handle to the pattern having the resource ID that you specify in the `patID` parameter. The `GetPattern` function calls the following Resource Manager function with these parameters:

```
GetResource('PAT ', patID);
```

If a pattern resource with the ID that you request does not exist, the `GetPattern` function returns `NIL`.

The data structure of type `PatHandle` is defined as follows:

```
TYPE  PatPrt      = ^Pattern;
      PatHandle   = ^PatPtr;
      Pattern = PACKED ARRAY[0..7] OF 0..255;
```

When you are finished using the pattern, dispose of its handle with the Memory Manager function `DisposeHandle`.

SPECIAL CONSIDERATIONS

The `GetPattern` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

SEE ALSO

The pattern resource is described on page 3-140; the `Pattern` record is described on page 3-40. See the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information about resources, the Resource Manager, and the `GetResource` function. See *Inside Macintosh: Memory* for information about the `DisposeHandle` procedure.

GetIndPattern

To get a pattern stored in a pattern list (`'PAT#'`) resource, you can use the `GetIndPattern` procedure.

```
PROCEDURE GetIndPattern (VAR thePattern: Pattern;
                         patListID: Integer; index: Integer);
```

`thePattern` A `Pattern` record.

`patListID` The resource ID for a resource of type `'PAT#'`.

`index` The index number for the desired pattern within the pattern list (`'PAT#'`) resource.

DESCRIPTION

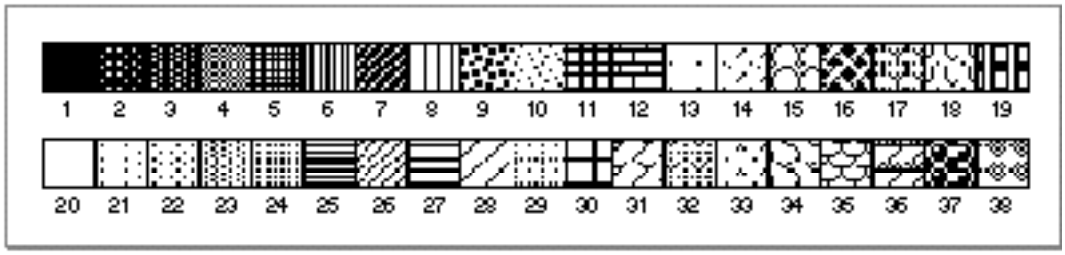
In the parameter `thePattern`, the `GetIndPattern` procedure returns a `Pattern` record for a pattern stored in a pattern list (`'PAT#'`) resource. Specify the resource ID for a pattern list (`'PAT#'`) resource in the `patListID` parameter. In the `index` parameter, specify the index number to a particular pattern stored in that resource. The index number can range from 1 to the number of patterns in the pattern list resource. The `GetIndPattern` procedure calls the following Resource Manager function with these parameters:

```
GetResource('PAT ', patListID);
```

There is a pattern list resource in the System file that contains the standard Macintosh patterns used by MacPaint. Figure 3-28 shows these standard patterns. The resource ID, and the constant you can use to represent it, are

```
CONST sysPatListID = 0;
```

Figure 3-28 Standard patterns



SPECIAL CONSIDERATIONS

The `GetIndPattern` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

The pattern list resource is described on page 3-141; the `Pattern` record is described on page 3-40. See the chapter “Resource Manager” in *Inside Macintosh: More Macintosh Toolbox* for more information about resources, the Resource Manager, and the `GetResource` function.

Customizing QuickDraw Operations

For each shape that QuickDraw can draw, there are procedures that perform these basic graphics operations on the shape: frame, paint, erase, invert, and fill. Those procedures in turn call a low-level drawing routine for the shape. For example, the `FrameOval`, `PaintOval`, `EraseOval`, `InvertOval`, and `FillOval` procedures all call the low-level procedure `StdOval`, which draws the oval.

Other low-level routines defined by QuickDraw are:

- The procedure called by `CopyBits` that performs bit and pixel transfer.
- The function that measures the width of text and is called by the QuickDraw text routines `CharWidth`, `StringWidth`, and `TextWidth`. (These QuickDraw text routines are described in the chapter "QuickDraw Text" in *Inside Macintosh: Text*.)
- The procedure that processes picture comments. The standard procedure ignores picture comments. (Picture comments are described in Appendix B.)
- The procedure that saves drawing commands as the definition of a picture, and the procedure that retrieves them. These two enable your application to draw on remote devices, print to the disk, get picture input from the disk, and support large pictures.

For each type of object QuickDraw can draw, including text and lines, there's a pointer to one of these low-level routines.

The `grafProcs` field of a `GrafPort` or `CGrafPort` record determines which low-level routines are called. If that field contains the value of `NIL`, the standard routines are called. You can set the `grafProcs` field to point to a record of pointers to your own routines. For a basic graphics port, this record of pointers is defined by a `QDProcs` record. As described in the chapter "Color QuickDraw," these pointers are contained in a `CQDProcs` record for a color graphics port. By changing these pointers, you can install your own routines, and either completely override the standard ones or call them after your routines have modified their parameters as necessary.

To assist you in setting up a record, basic QuickDraw provides the `SetStdProcs` procedure, which is described in the next section. You can use the `SetStdProcs` procedure to get a `QDProcs` record with fields that point to basic QuickDraw's standard low-level routines. You can then reset the routines with which you are concerned. By pointing to your modified `QDProcs` record in the `grafProcs` field of a `GrafPort` record, you can replace some or all of basic QuickDraw's standard low-level routines.

The standard QuickDraw low-level routines are described in the rest of this section. These low-level routines should be called only from your customized routines.

SetStdProcs

You can use the `SetStdProcs` procedure to get a `QDProcs` record with fields that point to basic QuickDraw's standard low-level routines. You can replace these low-level routines with your own, and then point to your modified `QDProcs` record in the `grafProcs` field of a `GrafPort` record to change basic QuickDraw's standard low-level behavior.

```
PROCEDURE SetStdProcs (VAR procs: QDProcs);
```

`procs` Upon completion, a `QDProcs` record with fields that point to basic QuickDraw's standard low-level routines.

DESCRIPTION

In the `procs` parameter, the `SetStdProcs` procedure returns a `QDProcs` record with fields that point to the standard low-level routines. You can change one or more fields of this record to point to your own routines and then set the basic graphics port to use this modified `QDProcs` record.

The routines you install in this `QDProcs` record must have the same calling sequences as the standard routines, which are described in the rest of this section.

SPECIAL CONSIDERATIONS

The Color QuickDraw procedure `SetStdCProcs` is analogous to the `SetStdProcs` procedure, which you should use with computers that support only basic QuickDraw. When drawing in a color graphics port, your application must always use `SetStdCProcs` instead of `SetStdProcs`.

SEE ALSO

The data structure of type `QDProcs` is described on page 3-39. The `SetStdCProcs` procedure is described in the chapter "Color QuickDraw."

The chapter "Pictures" in this book describes how to replace the low-level routines that read and write pictures.

StdText

The `StdText` procedure is QuickDraw's standard low-level routine for drawing text.

```
PROCEDURE StdText (byteCount: Integer; textBuf: Ptr;
                   numer,denom: Point);
```

`byteCount`	The number of bytes of text to draw.
`textBuf`	A memory structure containing the text to draw.
`numer`	Scaling numerator.
`denom`	Scaling denominator.

DESCRIPTION

The `StdText` procedure draws text from the arbitrary structure in memory specified by the `textBuf` parameter, starting from the first byte and continuing for the number of bytes specified in the `byteCount` parameter. The `numer` and `denom` parameters specify the scaling factor: `numer.v` over `denom.v` gives the vertical scaling, and `numer.h` over `denom.h` gives the horizontal scaling factor.

SPECIAL CONSIDERATIONS

The `StdText` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

QuickDraw's text-drawing capabilities are described in the chapter "QuickDraw Text" in *Inside Macintosh: Text*.

StdLine

The `StdLine` procedure is QuickDraw's standard low-level routine for drawing a line.

```
PROCEDURE StdLine (newPt: Point);
```

`newPt` The point to which to draw the line.

DESCRIPTION

The `StdLine` procedure draws a line from the current pen location to the location (in local coordinates) specified in the `newPt` parameter.

SPECIAL CONSIDERATIONS

The `StdLine` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

StdRect

The `StdRect` procedure is QuickDraw's standard low-level routine for drawing a rectangle.

```
PROCEDURE StdRect (verb: GrafVerb; r: Rect);
```

`verb` One of the following actions to perform, as defined for the `GrafVerb` data type:

```
GrafVerb = (frame, paint, erase, invert, fill);
```

`r` The rectangle to draw.

DESCRIPTION

The `StdRect` procedure draws the rectangle specified in the `r` parameter according to the action specified in the `verb` parameter.

SPECIAL CONSIDERATIONS

The `StdRect` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

StdRRect

The `StdRRect` procedure is QuickDraw's standard low-level routine for drawing a rounded rectangle.

```
PROCEDURE StdRRect (verb: GrafVerb; r: Rect;
                    ovalwidth,ovalHeight: Integer)
```

`verb` One of the following actions to perform, as defined for the `GrafVerb` data type:

```
GrafVerb = (frame, paint, erase, invert, fill);
```

`r` The rectangle to draw.

`ovalwidth` The width diameter for the corner oval.

`ovalHeight` The height diameter for the corner oval.

DESCRIPTION

The `StdRRect` procedure draws the rounded rectangle specified in the `r` parameter according to the action specified in the `verb` parameter. The `ovalWidth` and `ovalHeight` parameters specify the diameters of curvature for the corners.

SPECIAL CONSIDERATIONS

The `StdRRect` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

StdOval

The `StdOval` procedure is QuickDraw's standard low-level routine for drawing an oval.

```
PROCEDURE StdOval (verb: GrafVerb; r: Rect);
```

`verb` One of the following actions to perform, as defined for the `GrafVerb` data type:

```
GrafVerb = (frame, paint, erase, invert, fill);
```

`r` The rectangle to contain the oval.

DESCRIPTION

The `StdOval` procedure draws an oval inside the given rectangle specified in the `r` parameter according to the action specified in the `verb` parameter.

SPECIAL CONSIDERATIONS

The `StdOval` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

StdArc

The `StdArc` procedure is QuickDraw's standard low-level routine for drawing an arc or a wedge.

```
PROCEDURE StdArc (verb: GrafVerb; r: Rect;
                  startAngle,arcAngle: Integer);
```

`verb` One of the following actions to perform, as defined for the `GrafVerb` data type:

```
GrafVerb = (frame, paint, erase, invert, fill);
```

`r` The rectangle to contain the arc.

`startAngle` The beginning angle.

`arcAngle` The ending angle.

DESCRIPTION

Using the action specified in the `verb` parameter, the `StdArc` procedure draws an arc or wedge of the oval that fits inside the rectangle specified in the `r` parameter. The arc or wedge is bounded by the radii specified in the `startAngle` and `arcAngle` parameters. (The `startAngle` and `arcAngle` parameters are illustrated in Figure 3-20 on page 3-72.)

SPECIAL CONSIDERATIONS

The `StdArc` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

StdPoly

The `StdPoly` procedure is QuickDraw's standard low-level routine for drawing a polygon.

```
PROCEDURE StdPoly (verb: GrafVerb; poly: PolyHandle);
```

`verb` One of the following actions to perform, as defined for the `GrafVerb` data type:

```
GrafVerb = (frame, paint, erase, invert, fill);
```

`poly` A handle to the polygon data.

DESCRIPTION

The `StdPoly` procedure draws the polygon specified in the `poly` parameter according to the action specified in the `verb` parameter.

SPECIAL CONSIDERATIONS

The `StdPoly` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

StdRgn

The `StdRgn` procedure is QuickDraw's standard low-level routine for drawing a region.

```
PROCEDURE StdRgn (verb: GrafVerb; rgn: RgnHandle);
```

`verb` One of the following actions to perform, as defined for the `GrafVerb` data type:

```
GrafVerb = (frame, paint, erase, invert, fill);
```

`rgn` A handle to the region data.

DESCRIPTION

The `StdRgn` procedure draws the region specified in the `rgn` parameter according to the action specified in the `verb` parameter.

SPECIAL CONSIDERATIONS

The `StdRgn` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

StdBits

The `StdBits` procedure is QuickDraw's standard low-level routine for doing bit and pixel transfer.

```
PROCEDURE StdBits (VAR srcBits: BitMap; VAR srcRect,dstRect: Rect;
                   mode: Integer; maskRgn: RgnHandle);
```

`srcBits`	A bitmap or pixel map containing the image to copy.
`srcRect`	The source rectangle.
`dstRect`	The destination rectangle.
`mode`	The source mode for the copy.
`maskRgn`	A handle to a region acting as a mask for the transfer.

DESCRIPTION

The `StdBits` procedure transfers a bit or pixel image between the bitmap or pixel map specified in the `srcBits` parameter and bitmap of the current graphics port, just as if the `CopyBits` procedure were called with the same parameters and with a destination bitmap equal to `thePort^.portBits`.

SPECIAL CONSIDERATIONS

The `StdBits` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

See the description of the `CopyBits` procedure beginning on page 3-112 for a discussion of the destination bitmap and of the `srcBits`, `srcRect`, `dstRect`, `mode`, and `maskRgn` parameters.

StdComment

The `StdComment` procedure is QuickDraw's standard low-level routine for processing a picture comment.

```
PROCEDURE StdComment (kind,dataSize: Integer;
                      dataHandle: Handle);
```

`kind` The type of comment. See Appendix A in this book for a list of the standard constants (and the values they represent) used to specify common picture comment types.

`dataSize` The size of additional data.

`dataHandle` A handle to additional data.

DESCRIPTION

The `kind` parameter identifies the type of comment. The `dataHandle` parameter takes a handle to additional data, and the `dataSize` parameter specifies the size of that data in bytes. If there's no additional data for the comment, the value of the `dataHandle` parameter is `NIL` and the value of the `dataSize` parameter is 0. The `StdComment` procedure simply ignores the comment.

SPECIAL CONSIDERATIONS

The `StdComment` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

Picture comments are described in detail in Appendix B, "Using Picture Comments for Printing."

StdTxtMeas

The `StdTxtMeas` function is QuickDraw's standard low-level routine for measuring text width.

```
FUNCTION StdTxtMeas (byteCount: Integer; textAddr: Ptr;
                     VAR numer,denom: Point;
                     VAR info: FontInfo): Integer;
```

`byteCount`	The number of text bytes to measure.
`textAddr`	A pointer to the memory structure containing the text.
`numer`	Scaling numerator.
`denom`	Scaling denominator.
`info`	A `FontInfo` record.

DESCRIPTION

The `StdTxtMeas` function returns the width of the text stored in the arbitrary structure in memory specified by `textAddr`, starting with the first byte and continuing for `byteCount` bytes. The `numer` and `denom` parameters specify the scaling as in the `StdText` procedure; note that `StdTxtMeas` may change them.

SPECIAL CONSIDERATIONS

The `StdTxtMeas` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

SEE ALSO

QuickDraw's text-drawing capabilities are described in the chapter "QuickDraw Text" in *Inside Macintosh: Text*.

StdGetPic

The `StdGetPic` procedure is QuickDraw's standard low-level routine for retrieving information from the definition of a picture.

```
PROCEDURE StdGetPic (dataPtr: Ptr; byteCount: Integer);
```

`dataPtr`	A pointer to the collected picture data.
`byteCount`	The size of the picture data.

DESCRIPTION

The `StdGetPic` procedure retrieves from the definition of the currently open picture the next number of bytes as specified in the `byteCount` parameter. The `StdGetPic` procedure stores them in the data structure pointed to by the `dataPtr` parameter.

SEE ALSO

Pictures are described in the chapter "Pictures," which also provides a code sample illustrating how you can supply your application with its own low-level procedure for retrieving pictures.

StdPutPic

The `StdPutPic` procedure is QuickDraw's standard low-level routine for saving information as the definition of a picture.

```
PROCEDURE StdPutPic (dataPtr: Ptr; byteCount: Integer);
```

`dataPtr` A pointer to the collected picture data.

`byteCount` The size of the picture data.

DESCRIPTION

The `StdPutPic` procedure saves as the definition of the currently open picture the drawing commands stored in the data structure pointed to by the `dataPtr` parameter, starting with the first byte and continuing for the next number of bytes as specified in the `byteCount` parameter.

SPECIAL CONSIDERATIONS

The `StdPutPic` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

Pictures are described in the chapter "Pictures," which also provides a code sample illustrating how you can supply your application with its own low-level procedure for saving pictures.

Resources

This section describes the resources you can create to define bit patterns for use when drawing, painting, or filling. The pattern (`'PAT '`) resource defines a single bit pattern. The pattern list (`'PAT#'`) resource defines an array of bit patterns.

A bit pattern is a 64-bit image, organized as an 8-by-8 pixel square, that defines a repeating design or tone. (Resources for color pixel patterns are described in the chapter "Color QuickDraw.")

This section describes the structures of these resources after they are compiled by the Rez resource compiler, available from APDA. To create pattern and pattern list resources, you typically use a high-level tool such as the ResEdit application. You can then use the DeRez decompiler to convert your resources into Rez input when necessary.

The Pattern Resource

You can use a pattern resource to define a single bit pattern. A pattern resource is a resource of type `'PAT '`. All pattern resources that you create must have resource ID numbers greater than 128.

To retrieve the bit pattern stored in a pattern resource, you can use the `GetPattern` function, which is described on page 3-126. You can then specify that bit pattern for a fill pattern, background pattern, or pen pattern.

A pattern resource is defined to be of type `hex String[8]`; every bit represents a pixel in the 8-by-8 pixel pattern. If you examine the compiled version of a pattern resource, as represented in Figure 3-29, you find that it contains 8 bytes of information that define the 8-by-8 pixel square of the pattern.

Figure 3-29 Format of a compiled pattern (`'PAT '`) resource

The Pattern List Resource

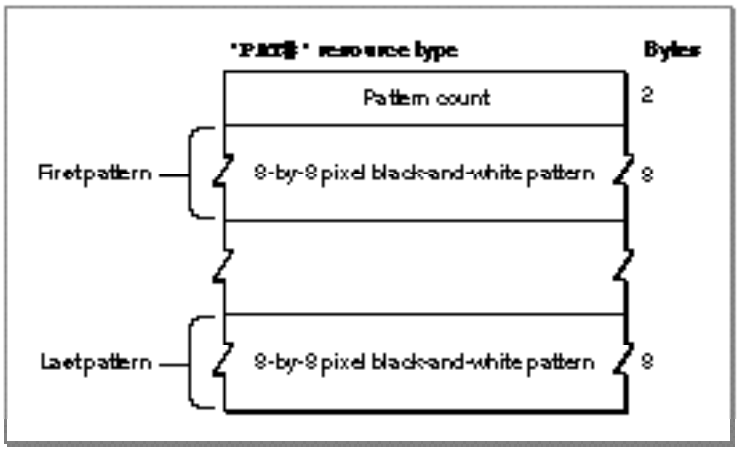
You can use a pattern list resource to define an array of bit patterns. A pattern list resource is a resource of type `'PAT#'`. All pattern list resources that you create must have resource ID numbers greater than 128.

To retrieve one of the bit patterns stored in a pattern list resource, you can use the `GetIndPattern` procedure, which is described on page 3-127. You can then specify that bit pattern for a fill pattern, background pattern, or pen pattern.

If you examine the compiled version of a pattern list resource, as represented in Figure 3-30, you find that it contains the following information:

- Pattern count. This is the number of bit patterns defined in this resource.
- An array of bit patterns, each of which contains 8 bytes of information that define the 8-by-8 pixel square of the pattern.

Figure 3-30 Format of a compiled pattern list (`'PAT#'`) resource



Summary of QuickDraw Drawing

Pascal Summary

Constants

```
CONST
   {basic QuickDraw colors}
   whiteColor     = 30;
   blackColor     = 33;
   yellowColor    = 69;
   magentaColor   = 137;
   redColor       = 205;
   cyanColor      = 273;
   greenColor     = 341;
   blueColor      = 409;

   {source modes for basic graphics ports}
   srcCopy        = 0;  {where source pixel is black, force destination }
                        { pixel black; where source pixel is white, force }
                        { destination pixel white}
   srcOr          = 1;  {where source pixel is black, force destination }
                        { pixel black; where source pixel is white, leave }
                        { destination pixel unaltered}
   srcXor         = 2;  {where source pixel is black, invert destination }
                        { pixel; where source pixel is white, leave }
                        { destination pixel unaltered}
   srcBic         = 3;  {where source pixel is black, force destination }
                        { pixel white; where source pixel is white, leave }
                        { destination pixel unaltered}
   notSrcCopy     = 4;  {where source pixel is black, force destination }
                        { pixel white; where source pixel is white, force }
                        { destination pixel black}
   notSrcOr       = 5;  {where source pixel is black, leave destination }
                        { pixel unaltered; where source pixel is white, }
                        { force destination pixel black}
```

```
notSrcXor      = 6;  {where source pixel is black, leave destination }
                     { pixel unaltered; where source pixel is white, }
                     { invert destination pixel}
notSrcBic      = 7;  {where source pixel is black, leave destination }
                     { pixel unaltered; where source pixel is white, }
                     { force destination pixel white}

{pattern modes}
patCopy        = 8;  {where pattern pixel is black, apply foreground }
                     { color to destination pixel; where pattern pixel }
                     { is white, apply background color to destination }
                     { pixel}
patOr          = 9;  {where pattern pixel is black, invert destination }
                     { pixel; where pattern pixel is white, leave }
                     { destination pixel unaltered}
patXor         = 10; {where pattern pixel is black, invert destination }
                     { pixel; where pattern pixel is white, leave }
                     { destination pixel unaltered}
patBic         = 11; {where pattern pixel is black, apply background }
                     { color to destination pixel; where pattern pixel }
                     { is white, leave destination pixel unaltered}
notPatCopy     = 12; {where pattern pixel is black, apply background }
                     { color to destination pixel; where pattern pixel }
                     { is white, apply foreground color to destination }
                     { pixel}
notPatOr       = 13; {where pattern pixel is black, leave destination }
                     { pixel unaltered; where pattern pixel is white, }
                     { apply foreground color to destination pixel}
notPatXor      = 14; {where pattern pixel is black, leave destination }
                     { pixel unaltered; where pattern pixel is white, }
                     { invert destination pixel}
notPatBic      = 15; {where pattern pixel is black, leave destination }
                     { pixel unaltered; where pattern pixel is white, }
                     { apply background color to destination pixel}

ditherCopy     = 64; {add to source mode for dithering}

{pattern list resource ID for patterns in the System file}
sysPatListID = 0;
```

Data Types

```
TYPE PolyPtr   = ^Polygon;
PolyHandle     = ^PolyPtr;
Polygon        =
RECORD
   polySize:   Integer;               {size in bytes}
   polyBBox:   Rect;                  {bounding rectangle}
   polyPoints: ARRAY[0..0] OF Point;  {vertices for polygon}
END;

PenState =
RECORD
   pnLoc:   Point;      {pen location}
   pnSize:  Point;      {pen size}
   pnMode:  Integer;    {pen's pattern mode}
   pnPat:   Pattern;    {pen pattern}
END;

QDProcsPtr     = ^QDProcs;
QDProcs        =
RECORD
   textProc:      Ptr;  {text drawing}
   lineProc:      Ptr;  {line drawing}
   rectProc:      Ptr;  {rectangle drawing}
   rRectProc:     Ptr;  {roundRect drawing}
   ovalProc:      Ptr;  {oval drawing}
   arcProc:       Ptr;  {arc/wedge drawing}
   rgnProc:       Ptr;  {region drawing}
   bitsProc:      Ptr;  {bit transfer}
   commentProc:   Ptr;  {picture comment processing}
   txMeasProc:    Ptr;  {text width measurement}
   getPicProc:    Ptr;  {picture retrieval}
   putPicProc:    Ptr;  {picture saving}
END;

GrafVerb = (frame,paint,erase,invert,fill);

PatPtr           = ^Pattern;
PatHandle        = ^PatPtr;
Pattern          = PACKED ARRAY[0..7] OF 0..255;
```

Routines

Managing the Graphics Pen

```
PROCEDURE HidePen;
PROCEDURE ShowPen;
PROCEDURE GetPen            (VAR pt: Point);
PROCEDURE GetPenState       (VAR pnState: PenState);
PROCEDURE SetPenState       (pnState: PenState);
PROCEDURE PenSize           (width,height: Integer);
PROCEDURE PenMode           (mode: Integer);
PROCEDURE PenPat            (pat: Pattern);
PROCEDURE PenNormal;
```

Changing the Background Bit Pattern

```
PROCEDURE BackPat           (pat: Pattern);
```

Drawing Lines

```
PROCEDURE MoveTo            (h,v: Integer);
PROCEDURE Move              (dh,dv: Integer);
PROCEDURE LineTo            (h,v: Integer);
PROCEDURE Line              (dh,dv: Integer);
```

Creating and Managing Rectangles

```
PROCEDURE SetRect           (VAR r: Rect; left,top,right,bottom: Integer);
PROCEDURE OffsetRect        (VAR r: Rect; dh,dv: Integer);
PROCEDURE InsetRect         (VAR r: Rect; dh,dv: Integer);
FUNCTION SectRect           (src1,src2: Rect; VAR dstRect: Rect): Boolean;
PROCEDURE UnionRect         (src1,src2: Rect; VAR dstRect: Rect);
FUNCTION PtInRect           (pt: Point; r: Rect): Boolean;
PROCEDURE Pt2Rect           (pt1,pt2: Point; VAR dstRect: Rect);
PROCEDURE PtToAngle         (r: Rect; pt: Point; VAR angle: Integer);
FUNCTION EqualRect          (rect1,rect2: Rect): Boolean;
FUNCTION EmptyRect          (r:  Rect): Boolean;
```

Drawing Rectangles

```
PROCEDURE FrameRect         (r: Rect);
PROCEDURE PaintRect         (r: Rect);
PROCEDURE FillRect          (r: Rect; pat: Pattern);
PROCEDURE EraseRect         (r: Rect);
PROCEDURE InvertRect        (r: Rect);
```

Drawing Rounded Rectangles

```
PROCEDURE FrameRoundRect    (r: Rect; ovalWidth,ovalHeight: Integer);
PROCEDURE PaintRoundRect    (r: Rect; ovalWidth,ovalHeight: Integer);
PROCEDURE FillRoundRect     (r: Rect; ovalWidth,ovalHeight: Integer;
                             pat: Pattern);
PROCEDURE EraseRoundRect    (r: Rect; ovalWidth,ovalHeight: Integer);
PROCEDURE InvertRoundRect   (r: Rect; ovalWidth,ovalHeight: Integer);
```

Drawing Ovals

```
PROCEDURE FrameOval         (r: Rect);
PROCEDURE PaintOval         (r: Rect);
PROCEDURE FillOval          (r: Rect; pat: Pattern);
PROCEDURE EraseOval         (r: Rect);
PROCEDURE InvertOval        (r: Rect);
```

Drawing Arcs and Wedges

```
PROCEDURE FrameArc          (r: Rect; startAngle,arcAngle: Integer);
PROCEDURE PaintArc          (r: Rect; startAngle,arcAngle: Integer);
PROCEDURE FillArc           (r: Rect; startAngle,arcAngle: Integer;
                             pat: Pattern);
PROCEDURE EraseArc          (r: Rect; startAngle,arcAngle: Integer);
PROCEDURE InvertArc         (r: Rect; startAngle,arcAngle: Integer);
```

Creating and Managing Polygons

```
FUNCTION OpenPoly           : PolyHandle;
PROCEDURE ClosePoly;
PROCEDURE OffsetPoly        (poly: PolyHandle; dh,dv: Integer);
PROCEDURE KillPoly          (poly: PolyHandle);
```

Drawing Polygons

```
PROCEDURE FramePoly         (poly: PolyHandle);
PROCEDURE PaintPoly         (poly: PolyHandle);
PROCEDURE FillPoly          (poly: PolyHandle; pat: Pattern);
PROCEDURE ErasePoly         (poly: PolyHandle);
PROCEDURE InvertPoly        (poly: PolyHandle);
```

Creating and Managing Regions

```
FUNCTION NewRgn             : RgnHandle;
PROCEDURE OpenRgn;
PROCEDURE CloseRgn          (dstRgn: rgnHandle);
PROCEDURE DisposeRgn        (rgn: RgnHandle);
PROCEDURE CopyRgn           (srcRgn,dstRgn: RgnHandle);
PROCEDURE SetEmptyRgn       (rgn: RgnHandle);
PROCEDURE SetRectRgn        (rgn: RgnHandle;
                             left,top,right,bottom: Integer);
PROCEDURE RectRgn           (rgn: RgnHandle; r: Rect);
PROCEDURE OffsetRgn         (rgn: RgnHandle; dh,dv: Integer);
PROCEDURE InsetRgn          (rgn: RgnHandle; dh,dv: Integer);
PROCEDURE SectRgn           (srcRgnA,srcRgnB,dstRgn: RgnHandle);
PROCEDURE UnionRgn          (srcRgnA,srcRgnB,dstRgn: RgnHandle);
PROCEDURE DiffRgn           (srcRgnA,srcRgnB,dstRgn: RgnHandle);
PROCEDURE XorRgn            (srcRgnA,srcRgnB,dstRgn: RgnHandle);
FUNCTION PtInRgn            (pt: Point; rgn: RgnHandle): Boolean;
FUNCTION RectInRgn          (r: Rect; rgn: RgnHandle): Boolean;
FUNCTION EqualRgn           (rgnA,rgnB: RgnHandle): Boolean;
FUNCTION EmptyRgn           (rgn: RgnHandle): Boolean;
```

Drawing Regions

```
PROCEDURE FrameRgn          (rgn: RgnHandle);
PROCEDURE PaintRgn          (rgn: RgnHandle);
PROCEDURE FillRgn           (rgn: RgnHandle; pat: Pattern);
PROCEDURE EraseRgn          (rgn: RgnHandle);
PROCEDURE InvertRgn         (rgn: RgnHandle);
```

Scaling and Mapping Points, Rectangles, Polygons, and Regions

```
PROCEDURE ScalePt           (VAR pt: Point; srcRect,dstRect: Rect);
PROCEDURE MapPt             (VAR pt: Point; srcRect,dstRect: Rect);
PROCEDURE MapRect           (VAR r: Rect; srcRect,dstRect: Rect);
PROCEDURE MapRgn            (rgn: RgnHandle; srcRect,dstRect: Rect);
PROCEDURE MapPoly           (poly: PolyHandle; srcRect,dstRect: Rect);
```

Calculating Black-and-White Fills

```
PROCEDURE SeedFill          (srcPtr,dstPtr: Ptr;
                             srcRow,dstRow,height,words,
                             seedH,seedV: Integer);
PROCEDURE CalcMask          (srcPtr,dstPtr: Ptr;
                             srcRow,dstRow,height,words: Integer);
```

Copying Images

```
PROCEDURE CopyBits          (srcBits,dstBits: BitMap;
                             srcRect,dstRect: Rect; mode: Integer;
                             maskRgn: RgnHandle);
PROCEDURE CopyMask          (srcBits,maskBits,dstBits: BitMap;
                             srcRect,maskRect,dstRect: Rect);
PROCEDURE CopyDeepMask      (srcBits: BitMap; maskBits: BitMap;
                             dstBits: BitMap; srcRect: Rect;
                             maskRect: Rect; dstRect: Rect;
                             mode: Integer; maskRgn: RgnHandle);
```

Drawing With the Eight-Color System

```
PROCEDURE ForeColor         (color: LongInt);
PROCEDURE BackColor         (color: LongInt);
PROCEDURE ColorBit          (whichBit: Integer);
```

Determining Whether QuickDraw Has Finished Drawing

```
FUNCTION QDDone             (port: GrafPtr): Boolean;
```

Getting Pattern Resources

```
FUNCTION GetPattern         (patID: Integer): PatHandle;
PROCEDURE GetIndPattern     (VAR thePattern: Pattern; patListID: Integer;
                             index: Integer);
```

Customizing QuickDraw Operations

```
PROCEDURE SetStdProcs       (VAR procs: QDProcs);
PROCEDURE StdText           (byteCount: Integer; textBuf: Ptr;
                             numer,denom: Point);
PROCEDURE StdLine           (newPt: Point);
PROCEDURE StdRect           (verb: GrafVerb; r: Rect);
PROCEDURE StdRRect          (verb: GrafVerb; r: Rect;
                             ovalwidth,ovalHeight: Integer);
PROCEDURE StdOval           (verb: GrafVerb; r: Rect);
PROCEDURE StdArc            (verb: GrafVerb; r: Rect;
                             startAngle,arcAngle: Integer);
PROCEDURE StdPoly           (verb: GrafVerb; poly: PolyHandle);
PROCEDURE StdRgn            (verb: GrafVerb; rgn: RgnHandle);
PROCEDURE StdBits           (VAR srcBits: BitMap;
                             VAR srcRect,dstRect: Rect; mode: Integer;
                             maskRgn: RgnHandle);
PROCEDURE StdComment        (kind,dataSize: Integer; dataHandle: Handle);
FUNCTION StdTxtMeas         (byteCount: Integer; textAddr: Ptr;
                             VAR numer, denom: Point;
                             VAR info: FontInfo): Integer;
PROCEDURE StdGetPic         (dataPtr: Ptr; byteCount: Integer);
PROCEDURE StdPutPic         (dataPtr: Ptr; byteCount: Integer);
```

C Summary

Constants

```
enum {
   /* basic QuickDraw colors */
   whiteColor     = 30;
   blackColor     = 33;
   yellowColor    = 69;
   magentaColor   = 137;
   redColor       = 205;
   cyanColor      = 273;
   greenColor     = 341;
   blueColor      = 409;
```

```
/* source modes */
srcCopy         = 0,  /* where source pixel is black, force destination
                         pixel black; where source pixel is white, force
                         destination pixel white */
srcOr           = 1,  /* where source pixel is black, force destination
                         pixel black; where source pixel is white, leave
                         destination pixel unaltered */
srcXor          = 2,  /* where source pixel is black, invert destination
                         pixel; where source pixel is white, leave
                         destination pixel unaltered */
srcBic          = 3,  /* where source pixel is black, force destination
                         pixel white; where source pixel is white, leave
                         destination pixel unaltered */
notSrcCopy      = 4,  /* where source pixel is black, force destination
                         pixel white; where source pixel is white, force
                         destination pixel black */
notSrcOr        = 5,  /* where source pixel is black, leave destination
                         pixel unaltered; where source pixel is white,
                         force destination pixel black */
notSrcXor       = 6,  /* where source pixel is black, leave destination
                         pixel unaltered; where source pixel is white,
                         invert destination pixel*/
notSrcBic       = 7,  /* where source pixel is black, leave destination
                         pixel unaltered; where source pixel is white,
                         force destination pixel white */

/* pattern modes */
patCopy         = 8,  /* where pattern pixel is black, apply foreground
                         color to destination pixel; where pattern pixel
                         is white, apply background color to destination
                         pixel */
patOr           = 9,  /* where pattern pixel is black, invert destination
                         pixel; where pattern pixel is white, leave
                         destination pixel unaltered */
patXor          = 10; /* where pattern pixel is black, invert destination
                         pixel; where pattern pixel is white, leave
                         destination pixel unaltered */
patBic          = 11; /* where pattern pixel is black, apply background
                         color to destination pixel; where pattern pixel
                         is white, leave destination pixel unaltered */
```

```
   notPatCopy     = 12; /* where pattern pixel is black, apply background
                           color to destination pixel; where pattern pixel
                           is white, apply foreground color to destination
                           pixel */
   notPatOr       = 13; /* where pattern pixel is black, leave destination
                           pixel unaltered; where pattern pixel is white,
                           apply foreground color to destination pixel */
   notPatXor      = 14; /* where pattern pixel is black, leave destination
                           pixel unaltered; where pattern pixel is white,
                           invert destination pixel */
   notPatBic      = 15; /* where pattern pixel is black, leave destination
                           pixel unaltered; where pattern pixel is white,
                           apply background color to destination pixel */

   ditherCopy     = 64, /* add to source mode for dithering */

   /* pattern list resource ID for patterns in the System file */
   sysPatListID = 0
};
```

Data Types

```
struct Polygon {
   short polySize;      /* size in bytes */
   Rect  polyBBox;      /* bounding rectangle */
   Point polyPoints[1]; /* vertices for polygon */
};
typedef struct Polygon Polygon;
typedef Polygon *PolyPtr, **PolyHandle;

struct PenState {
   Point    pnLoc;      /* pen location */
   Point    pnSize;     /* pen size */
   short    pnMode;     /* pen's pattern mode */
   Pattern  pnPat;      /* pen pattern */
};
typedef struct PenState PenState;

struct QDProcs {
   Ptr textProc;        /* text drawing */
   Ptr lineProc;        /* line drawing */
   Ptr rectProc;        /* rectangle drawing */
   Ptr rRectProc;       /* roundRect drawing */
   Ptr ovalProc;        /* oval drawing */
```

```
   Ptr arcProc;         /* arc and wedge drawing */
   Ptr polyProc;        /* region drawing */
   Ptr rgnProc;         /* region drawing */
   Ptr bitsProc;        /* bit transfer */
   Ptr commentProc;     /* picture comment processing */
   Ptr txMeasProc;      /* text width measurement */
   Ptr getPicProc;      /* picture retrieval */
   Ptr putPicProc;      /* picture saving */
};
typedef struct QDProcs QDProcs;
typedef QDProcs *QDProcsPtr;

enum {frame,paint,erase,invert,fill};
typedef unsigned char GrafVerb;

struct Pattern{
   unsigned char pat[8];
};
typedef struct Pattern Pattern;
typedef Pattern *PatPtr;
typedef const unsigned char *ConstPatternParam;
typedef PatPtr *PatHandle;
```

Functions

Managing the Graphics Pen

```
pascal void HidePen         (void);
pascal void ShowPen         (void);
pascal void GetPen          (Point *pt);
pascal void GetPenState     (PenState *pnState);
pascal void SetPenState     (const PenState *pnState);
pascal void PenSize         (short width, short height);
pascal void PenMode         (short mode);
pascal void PenPat          (ConstPatternParam pat);
pascal void PenNormal       (void);
```

Changing the Background Bit Pattern

```
pascal void BackPat         (ConstPatternParam pat);
```

Drawing Lines

```
pascal void MoveTo          (short h, short v);
pascal void Move            (short dh, short dv);
pascal void LineTo          (short h, short v);
pascal void Line            (short dh, short dv);
```

Creating and Managing Rectangles

```
pascal void SetRect         (Rect *r, short left, short top, short right,
                             short bottom);
pascal void OffsetRect      (Rect *r, short dh, short dv);
pascal void InsetRect       (Rect *r, short dh, short dv);
pascal Boolean SectRect     (const Rect *src1, const Rect *src2,
                             Rect *dstRect);
pascal void UnionRect       (const Rect *src1, const Rect *src2,
                             Rect *dstRect);
pascal Boolean PtInRect     (Point pt, const Rect *r);
pascal void Pt2Rect         (Point pt1, Point pt2, Rect *dstRect);
pascal void PtToAngle       (const Rect *r, Point pt, short *angle);
pascal Boolean EqualRect    (const Rect *rect1, const Rect *rect2);
pascal Boolean EmptyRect    (const Rect *r);
```

Drawing Rectangles

```
pascal void FrameRect       (const Rect *r);
pascal void PaintRect       (const Rect *r);
pascal void FillRect        (const Rect *r, ConstPatternParam pat);
pascal void EraseRect       (const Rect *r);
pascal void InvertRect      (const Rect *r);
```

Drawing Rounded Rectangles

```
pascal void FrameRoundRect  (const Rect *r, short ovalWidth,
                             short ovalHeight);
pascal void PaintRoundRect  (const Rect *r, short ovalWidth,
                             short ovalHeight);
pascal void FillRoundRect   (const Rect *r, short ovalWidth,
                             short ovalHeight, ConstPatternParam pat);
```

```
pascal void EraseRoundRect  (const Rect *r, short ovalWidth,
                             short ovalHeight);
pascal void InvertRoundRect
                            (const Rect *r, short ovalWidth,
                             short ovalHeight);
```

Drawing Ovals

```
pascal void FrameOval       (const Rect *r);
pascal void PaintOval       (const Rect *r);
pascal void FillOval        (const Rect *r, ConstPatternParam pat);
pascal void EraseOval       (const Rect *r);
pascal void InvertOval      (const Rect *r);
```

Drawing Arcs and Wedges

```
pascal void FrameArc        (const Rect *r, short startAngle,
                             short arcAngle);
pascal void PaintArc        (const Rect *r, short startAngle,
                             short arcAngle);
pascal void FillArc         (const Rect *r, short startAngle,
                             short arcAngle, ConstPatternParam pat);
pascal void EraseArc        (const Rect *r, short startAngle,
                             short arcAngle);
pascal void InvertArc       (const Rect *r, short startAngle,
                             short arcAngle);
```

Creating and Managing Polygons

```
pascal PolyHandle OpenPoly  (void);
pascal void ClosePoly       (void);
pascal void OffsetPoly      (PolyHandle poly, short dh, short dv);
pascal void KillPoly        (PolyHandle poly);
```

Drawing and Painting Polygons

```
pascal void FramePoly       (PolyHandle poly);
pascal void PaintPoly       (PolyHandle poly);
pascal void FillPoly        (PolyHandle poly, ConstPatternParam pat);
pascal void ErasePoly       (PolyHandle poly);
pascal void InvertPoly      (PolyHandle poly);
```

Creating and Managing Regions

```
pascal RgnHandle NewRgn     (void);
pascal void OpenRgn         (void);
pascal void CloseRgn        (RgnHandle dstRgn);
pascal void DisposeRgn      (RgnHandle rgn);
pascal void CopyRgn         (RgnHandle srcRgn, RgnHandle dstRgn);
pascal void SetEmptyRgn     (RgnHandle rgn);
pascal void SetRectRgn      (RgnHandle rgn, short left, short top,
                             short right, short bottom);
pascal void RectRgn         (RgnHandle rgn, const Rect *r);
pascal void OffsetRgn       (RgnHandle rgn, short dh, short dv);
pascal void InsetRgn        (RgnHandle rgn, short dh, short dv);
pascal void SectRgn         (RgnHandle srcRgnA, RgnHandle srcRgnB,
                             RgnHandle dstRgn);
pascal void UnionRgn        (RgnHandle srcRgnA, RgnHandle srcRgnB,
                             RgnHandle dstRgn);
pascal void DiffRgn         (RgnHandle srcRgnA, RgnHandle srcRgnB,
                             RgnHandle dstRgn);
pascal void XorRgn          (RgnHandle srcRgnA, RgnHandle srcRgnB,
                             RgnHandle dstRgn);
pascal Boolean PtInRgn      (Point pt, RgnHandle rgn);
pascal Boolean RectInRgn    (const Rect *r, RgnHandle rgn);
pascal Boolean EqualRgn     (RgnHandle rgnA, RgnHandle rgnB);
pascal Boolean EmptyRgn     (RgnHandle rgn);
```

Drawing Regions

```
pascal void FrameRgn        (RgnHandle rgn);
pascal void PaintRgn        (RgnHandle rgn);
pascal void FillRgn         (RgnHandle rgn, ConstPatternParam pat);
pascal void EraseRgn        (RgnHandle rgn);
pascal void InvertRgn       (RgnHandle rgn);
```

Scaling and Mapping Points, Rectangles, Polygons, and Regions

```
pascal void ScalePt         (Point *pt, const Rect *srcRect,
                             const Rect *dstRect);
pascal void MapPt           (Point *pt, const Rect *srcRect,
                             const Rect *dstRect);
pascal void MapRect         (Rect *r, const Rect *srcRect,
                             const Rect *dstRect);
```

```
pascal void MapRgn          (RgnHandle rgn, const Rect *srcRect,
                             const Rect *dstRect);
pascal void MapPoly         (PolyHandle poly, const Rect *srcRect,
                             const Rect *dstRect);
```

Calculating Black-and-White Fills

```
pascal void SeedFill        (const void *srcPtr, void *dstPtr,
                             short srcRow, short dstRow, short height,
                             short words, short seedH, short seedV);
pascal void CalcMask        (const void *srcPtr, void *dstPtr,
                             short srcRow, short dstRow, short height,
                             short words);
```

Copying Images

```
pascal void CopyBits        (const BitMap *srcBits,
                             const BitMap *dstBits, const Rect *srcRect,
                             const Rect *dstRect, short mode,
                             RgnHandle maskRgn);
pascal void CopyMask        (const BitMap *srcBits,
                             const BitMap *maskBits, const BitMap *dstBits,
                             const Rect *srcRect, const Rect *maskRect,
                             const Rect *dstRect);
pascal void CopyDeepMask    (const BitMap *srcBits, const BitMap *maskBits,
                             const BitMap *dstBits, const Rect *srcRect,
                             const Rect *maskRect, const Rect *dstRect,
                             short mode, RgnHandle maskRgn);
```

Drawing With the Eight-Color System

```
pascal void ForeColor       (long color);
pascal void BackColor       (long color);
pascal void ColorBit        (short whichBit);
```

Determining Whether QuickDraw Has Finished Drawing

```
pascal Boolean QDDone       (GrafPtr port);
```

Getting Pattern Resources

```
pascal PatHandle GetPattern
                            (short patternID);
pascal void GetIndPattern   (Pattern thePat, short patternListID,
                             short index);
```

Customizing QuickDraw Operations

```
pascal void SetStdProcs     (QDProcs *procs);
pascal void StdText         (short count, const void *textAddr,
                             Point numer, Point denom);
pascal void StdLine         (Point newPt);
pascal void StdRect         (GrafVerb verb, const Rect *r);
pascal void StdRRect        (GrafVerb verb, const Rect *r,
                             short ovalWidth, short ovalHeight);
pascal void StdOval         (GrafVerb verb, const Rect *r);
pascal void StdArc          (GrafVerb verb, const Rect *r,
                             short startAngle, short arcAngle);
pascal void StdPoly         (GrafVerb verb, PolyHandle poly);
pascal void StdRgn          (GrafVerb verb, RgnHandle rgn);
pascal void StdBits         (const BitMap *srcBits,
                             const Rect *srcRect, const Rect *dstRect,
                             short mode, RgnHandle maskRgn);
pascal void StdComment      (short kind, short dataSize, Handle dataHandle);
pascal short StdTxtMeas     (short byteCount, const void *textAddr,
                             Point *numer, Point *denom, FontInfo *info);
pascal void StdGetPic       (void *dataPtr, short byteCount);
pascal void StdPutPic       (const void *dataPtr, short byteCount);
```


Assembly-Language Summary

Data Structures

Polygon Data Structure

0	polySize	word	total bytes in this structure
2	polyBBox	8 bytes	bounding rectangle
10	polyPoints	variable	vertices, each consisting of a long (point)

PenState Data Structure

0	psLoc	long	pen location
4	psSize	long	pen size
8	psMode	word	pattern mode
10	psPat	8 bytes	pattern

QDProcs Data Structure

0	textProc	long	pointer to text-drawing routine
4	lineProc	long	pointer to line-drawing routine
8	rectProc	long	pointer to rectangle-drawing routine
12	rRectProc	long	pointer to rounded rectangle–drawing routine
16	ovalProc	long	pointer to oval-drawing routine
20	arcProc	long	pointer to arc/wedge-drawing routine
24	polyProc	long	pointer to polygon-drawing routine
28	rgnProc	long	pointer to region-drawing routine
32	bitsProc	long	pointer to bit transfer routine
36	commentProc	long	pointer to picture comment–processing routine
40	txMeasProc	long	pointer to text-width measurement routine
44	getPicProc	long	pointer to picture retrieval routine
48	putPicProc	long	pointer to picture-saving routine

Trap Macro Requiring Routine Selector

_QDExtensions

Selector	Routine
$00040013	QDDone

Global Variables

black	All-black pattern.
dkGray	75% gray pattern.
gray	50% gray pattern.
ltGray	25% gray pattern.
white	All-white pattern.