Color QuickDraw

Contents

This chapter describes Color QuickDraw, the version of QuickDraw that provides a range of color and grayscale capabilities to your application. You should read this chapter if your application needs to use shades of gray or more colors than the eight predefined colors provided by basic QuickDraw.

Read this chapter to learn how to set up and manage a **color graphics port**—the sophisticated drawing environment available on Macintosh computers that support Color QuickDraw. You should also read this chapter to learn how to draw using many more colors than are available with basic QuickDraw's eight-color system.

Color QuickDraw supports all of the routines described in the previous chapters of this book. For a color graphics port, for example, you can use the `ScrollRect` and `SetOrigin` procedures, which are described in the chapter "Basic QuickDraw." Furthermore, you can use the drawing routines described in the chapter "QuickDraw Drawing" to draw with the sophisticated color and grayscale capabilities available to color graphics ports. For example, after creating an `RGBColor` record that describes a medium shade of green, you can use the Color QuickDraw procedure `RGBForeColor` to make that color the foreground color. Then, when you use the `FrameRect` procedure, Color QuickDraw draws the outline for your rectangle with your specified shade of green.

To prevent the choppiness that can occur when you build a complex color image onscreen, your application typically should prepare the image in an offscreen graphics world and then copy it to an onscreen color graphics port as described in the chapter "Offscreen Graphics Worlds." If you want to optimize your application's drawing for screens with different color capabilities, see the chapter "Graphics Devices."

This chapter describes color graphics ports and Color QuickDraw's routines for drawing in color. For many applications, Color QuickDraw provides a device-independent interface: draw colors in the color graphics port for a window, and Color QuickDraw automatically manages the path to the screen. If your application needs more control over its color environment, Macintosh system software provides additional graphics managers to enhance your application's color-handling abilities. These managers are described in *Inside Macintosh: Advanced Color Imaging*, which shows you how to

- manage color selection across a variety of indexed devices by using the Palette Manager
- solicit color choices from users by using the Color Picker
- match colors between the screen and other devices—such as scanners and printers—by using the ColorSync Utilities
- directly manipulate the fields of the CLUT on an indexed device—although most applications should never need to do so—by using the Color Manager

About Color QuickDraw

Color QuickDraw is a collection of system software routines that your application can use to display hundreds, thousands, even millions of colors on capable screens. Color QuickDraw is available on all newer models of Macintosh computers; only those older computers based on the Motorola 68000 processor provide no support for Color QuickDraw.

Color QuickDraw performs its operations in a graphics port called a *color graphics port*, which is based on a data structure of type `CGrafPort`. As with basic graphics ports (which are based on a data structure of type `GrafPort`), each color graphics port has its own local coordinate system. All fields in a `CGrafPort` record are expressed in these coordinates, and all calculations and actions that Color QuickDraw performs use its local coordinate system.

As described in the chapter "QuickDraw Drawing," you can draw into a basic graphics port using eight predefined colors. With a color graphics port, however, you can define your own colors with which to draw. With Color QuickDraw, your application works in an abstract color space defined by three axes of red, green, and blue (RGB). Although the range of colors actually available to your application depends on the user's computer system, Color QuickDraw provides a consistent way for your application to deal with color, regardless of the characteristics of your user's screen and software configuration.

RGB Colors

When using Color QuickDraw, you specify colors as RGB colors. An RGB color is defined by its red, green, and blue components. For example, when each of the red, green, and blue components of a color is at maximum intensity ($FFFF), the result is the color white. When each of the components has zero intensity ($0000), the result is the color black.

You specify a color to Color QuickDraw by creating an `RGBColor` record in which you use three 16-bit unsigned integers to assign intensity values for the three additive primary colors. The `RGBColor` data type is defined as follows.

```
TYPE RGBColor =
RECORD
   red:     Integer;    {red component}
   green:   Integer;    {green component}
   blue:    Integer;    {blue component}
END;
```

When you specify an RGB color in an `RGBColor` record and then draw with that color, Color QuickDraw translates that color to the various indexed or direct devices that your user may be using.

For example, your application can use Color QuickDraw to display images containing up to 256 different colors on indexed devices. An **indexed device** is a graphics device—that is, a plug-in video card, a video interface built into a Macintosh computer, or an offscreen graphics world—that supports up to 256 colors in a color lookup table. Indexed devices support pixels of 1-bit, 2-bit, 4-bit, or 8-bit depths. On indexed devices, each pixel is represented in memory by an index to the graphics device's color lookup table (also known as the *CLUT*), where the currently available colors are stored. Such images, although limited in hue, take up relatively small amounts of memory. Color QuickDraw, working with the Color Manager, automatically matches the color your application specifies to the closest available color in the CLUT.

Your application can use the Palette Manager, described in the chapter "Palette Manager" in *Inside Macintosh: Advanced Color Imaging,* to exercise greater control of the colors in the CLUT. Note, however, that some Macintosh computers—such as black-and-white and grayscale PowerBook computers—have a fixed CLUT, which your application cannot change.

On direct devices, your application can use Color QuickDraw to display images containing thousands or millions of different colors. A **direct device** is a graphics device that supports up to 16 million colors having a direct correlation between a value placed in the graphics device and the color displayed onscreen. On attached direct devices, each pixel is represented in memory by the most significant bits of the actual red, green, and blue component values specified in an `RGBColor` record by your application.

Other output devices may render colors that differ from RGB colors; for example, many color printers work with CMYK (cyan, magenta, yellow, and black) colors. See *Inside Macintosh: Advanced Color Imaging* for information about color matching between screens, which use RGB colors, and devices—like printers—that use CMYK or other colors.

The Color Drawing Environment: Color Graphics Ports

A color graphics port defines a complete drawing environment that determines where and how color graphics operations take place. As with basic graphics ports, you can open many color graphics ports at once. Each color graphics port has its own local coordinate system, drawing pattern, background pattern, pen size and location, foreground color, background color, and pixel map. Using the `SetPort` procedure (described in the chapter "Basic QuickDraw"), or the `SetGWorld` procedure (described in the chapter "Offscreen Graphics Worlds"), you can instantly switch from one color or basic graphics port to another.

When you use Window Manager and Dialog Manager routines and resources to create color windows, dialog boxes, and alert boxes, these managers automatically create color graphics ports for you. As described in *Inside Macintosh: Macintosh Toolbox Essentials*, for example, a color graphics port is automatically created when you use the Window Manager function `GetNewCWindow` or `NewCWindow`. Color graphics ports are automatically created when your application provides the color-aware resources `'dctb'` and `'actb'` and then uses the Dialog Manager routines `GetNewDialog` and `Alert`.

A color graphics port is defined by a `CGrafPort` record, which is diagrammed in Figure 4-1. Some aspects of its contents are discussed after the figure; see page 4-48 for a complete description of the fields. Your application generally should not directly set any fields of a `CGrafPort` record; instead you should use the QuickDraw routines described in this book to manipulate them.

Figure 4-1 The color graphics port

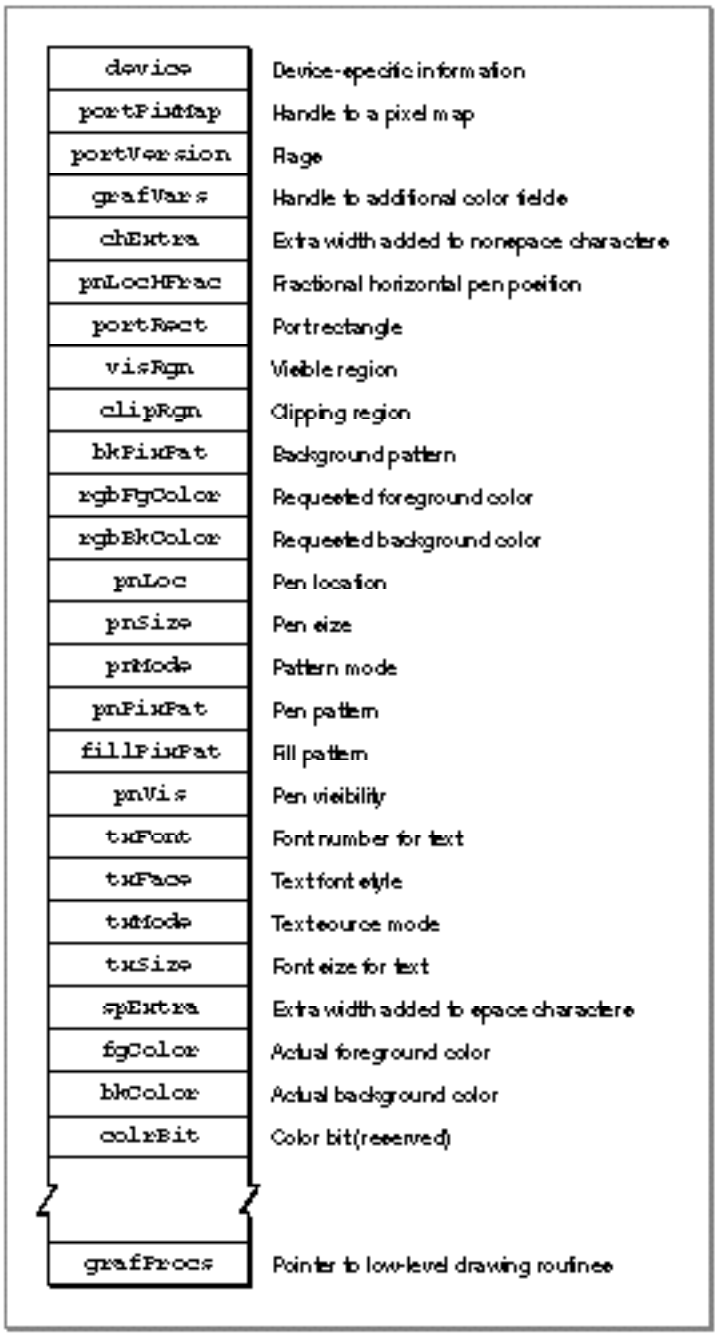


Table 4-3 on page 4-64 shows initial values for a `CGrafPort` record. A `CGrafPort` record is the same size as a `GrafPort` record (described in the chapter "Basic QuickDraw"), and most of the fields are identical for these two records. The important differences between these two data types are listed here:

- In a `GrafPort` record, the `portBits` field contains a complete 14-byte `BitMap` record. In a `CGrafPort` record, this field is partly replaced by the 4-byte `portPixMap` field; this field contains a handle to a `PixMap` record.
- In what would be the `rowBytes` field of the `BitMap` record stored in the `portBits` field of a `GrafPort` record, a `CGrafPort` record has a 2-byte `portVersion` field in which the 2 high bits are always set. QuickDraw uses these bits to distinguish `CGrafPort` records from `GrafPort` records, in which the 2 high bits of the `rowBytes` field are always clear.
- Following the `portVersion` field in the `CGrafPort` record is the `grafVars` field, which contains a handle to a `GrafVars` record; this handle is not included in a `GrafPort` record. The `GrafVars` record contains color information used by Color QuickDraw and the Palette Manager.
- In a `GrafPort` record, the `bkPat`, `pnPat`, and `fillPat` fields hold 8-byte bit patterns. In a `CGrafPort` record, these fields are partly replaced by three 4-byte handles to pixel patterns. The resulting 12 bytes of additional space are taken up by the `rgbFgColor` and `rgbBkColor` fields, which contain 6-byte `RGBColor` records specifying the optimal foreground and background colors for the color graphics port. Note that the closest matching available colors, which Color QuickDraw actually uses for the foreground and background, are stored in the `fgColor` and `bkColor` fields of the `CGrafPort` record.
- In a `GrafPort` record, you can supply the `grafProcs` field with a pointer to a `QDProcs` record that your application can store into if you want to customize QuickDraw drawing routines or use QuickDraw in other advanced, highly specialized ways. If you supply custom QuickDraw drawing routines in a `CGrafPort` record, you must provide this field with a pointer to a data structure of type `CQDProcs`.

Working with a `CGrafPort` record is much like using a `GrafPort` record. The routines `SetPort`, `GetPort`, `PortSize`, `SetOrigin`, `SetPortBits`, and `MovePortTo` operate on either port type, and the global variable `ThePort` points to the current graphics port no matter which type it is. (Remember that drawing always takes place in the current graphics port.) These routines are described in the chapter "Basic QuickDraw."

If you find it necessary, you can use type coercion to convert between `GrafPtr` and `CGrafPtr` records. For example:

```
VAR myPort: CGrafPtr;
SetPort (GrafPtr(myPort));
```

Note

You can use all QuickDraw drawing commands when drawing into a graphics port created with a `CGrafPort` record, and you can use all Color QuickDraw drawing commands (such as `FillCRect`) when drawing into a graphics port created with a `GrafPort` record. However, Color QuickDraw drawing commands used with a `GrafPort` record don't take advantage of Color QuickDraw's color features. ◆

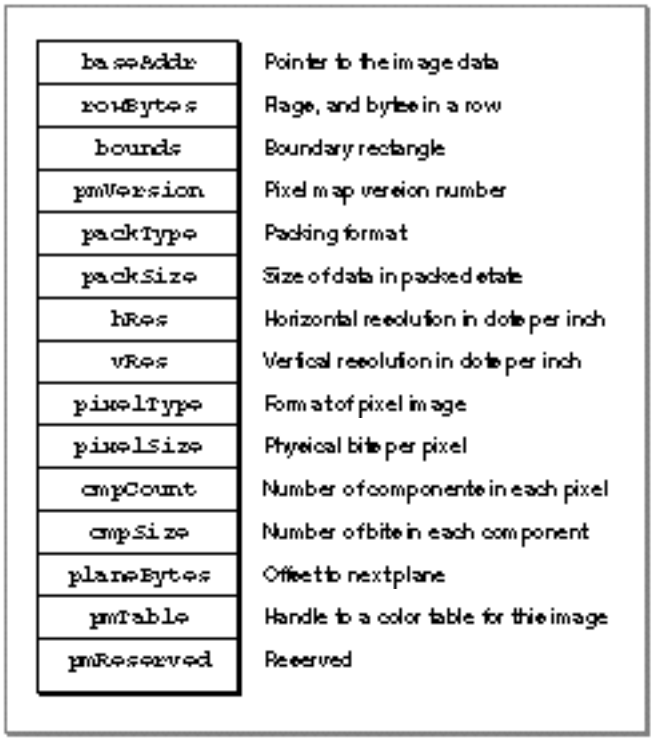
While the `CGrafPort` record contains information for a color window, there can be many windows on a screen, and even more than one screen. The `GDevice` record, described in the chapter "Graphics Devices," is the data structure that holds state information about a graphics device—such as the size of its boundary rectangle and whether the device is indexed or direct. Like the graphics port, the `GDevice` record is created automatically for you: QuickDraw uses information supplied by the Slot Manager to create a `GDevice` record for each graphics device found during startup. Many applications can let Color QuickDraw manage multiple screens of differing pixel depths. If your application needs more control over graphics device management—if your application needs certain screen depths to function effectively, for example—you can use the routines described in the chapter "Graphics Devices."

Pixel Maps

The `portPixMap` field of a `CGrafPort` record contains a handle to a **pixel map,** a data structure of type `PixMap`. Just as basic QuickDraw does all of its drawing in a bitmap, Color QuickDraw draws in a pixel map.

The representation of a color image in memory is a **pixel image,** analogous to the bit image used by basic QuickDraw. A `PixMap` record includes a pointer to a pixel image, its dimensions, storage format, depth, resolution, and color usage. The pixel map is diagrammed in Figure 4-2. Some aspects of its contents are discussed after the figure; see page 4-46 for a complete description of its fields.

Figure 4-2 The pixel map



The `baseAddr` field of a `PixMap` record contains a pointer to the beginning of the onscreen pixel image for a pixel map. The pixel image that appears on a screen is normally stored on a graphics card rather than in main memory. (There can be several pixel maps pointing to the same pixel image, each imposing its own coordinate system on it.)

As with a bitmap, the pixel map's boundary rectangle is initially set to the size of the main screen. However, you should never use a pixel map's boundary rectangle to determine the size of the screen; instead use the value of the `gdRect` field of the `GDevice` record for the screen, as described in the chapter "Graphics Devices" in this book.

The number of bits per pixel in the pixel image is called the **pixel depth.** Pixels on indexed devices can be 1, 2, 4, or 8 bits deep. (A pixel image that is 1 bit deep is equivalent to a bit image.) Pixels on direct devices can be 16 or 32 bits deep. (Even if your application creates a basic graphics port on a direct device, pixels are never less

than one of these two depths.) When a user uses the Monitors control panel to set a 16-bit or 32-bit direct device to use 2, 4, 16, or 256 colors as a grayscale or color device, the direct device creates a CLUT and operates like an indexed device.

When your application specifies an RGB color for some pixel in a pixel image, Color QuickDraw translates that color into a value appropriate for display on the user's screen; Color QuickDraw stores this value in the pixel. The **pixel value** is a number used by system software and a graphics device to represent a color. The translation from the color you specify in an `RGBColor` record to a pixel value is performed at the time you draw the color. The process differs for indexed and direct devices, as described here.

- When drawing on indexed devices, Color QuickDraw calls the Color Manager to supply the index to the color that most closely matches the requested color in the current device's CLUT. This index becomes the pixel value for that color.
- When drawing on direct devices, Color QuickDraw truncates the least significant bits from the `red`, `green`, and `blue` fields of the `RGBColor` record. This becomes the pixel value that Color QuickDraw sends to the graphics device.

This process is described in greater detail in "Color QuickDraw's Translation of RGB Colors to Pixel Values" beginning on page 4-13.

The `hRes` and `vRes` fields of the `PixMap` record describe the horizontal and vertical resolution of the image in pixels per inch, abbreviated as dpi (dots per inch). The values for these fields are of type `Fixed`; by default, the value for each is $00480000 (for 72 dpi), but Color QuickDraw supports `PixMap` records of other resolutions. For example, `PixMap` records for scanners and frame grabbers can have dpi resolutions of 150, 200, 300, or greater.

The `pixelType` field of the `PixMap` record specifies the format—indexed or direct—used to hold the pixels in the image. For indexed devices the value is 0; for direct devices it is 16 (which can be represented by the constant `RGBDirect`).

The `pixelSize` field specifies the pixel depth. Indexed devices can be 1, 2, 4, or 8 bits deep; direct devices can be 16 or 32 bits deep.

The `cmpCount` and `cmpSize` fields describe how the pixel values are organized. For pixels on indexed devices, the color component count (stored in the `cmpCount` field) is 1—for the index into the graphics device's CLUT, where the colors are stored. For pixels on direct devices, the color component count is 3—for the red, green, and blue components of each pixel.

The `cmpSize` field specifies how large each color component is. For indexed devices it is the same value as that in the `pixelSize` field: 1, 2, 4, or 8 bits. For direct pixels, each of the three color components can be either 5 bits for a 16-bit pixel (1 of these 16 bits is unused), or 8 bits for a 32-bit pixel (8 of these 32 bits are unused).

The `planeBytes` field specifies an offset in bytes from one plane to another. Since Color QuickDraw doesn't support multiple-plane images, the value of this field is always 0.

Finally, the `pmTable` field contains a handle to the `ColorTable` record. **Color tables** define the colors available for pixel images on indexed devices. (The Color Manager stores a color table for the currently available colors in the graphics device's CLUT; you can use the Palette Manager to assign different color tables to your different windows.)

You can create color tables using either `ColorTable` records (described on page 4-56) or color table (`'clut'`) resources (described on page 4-104). Pixel images on direct devices don't need a color table because the colors are stored right in the pixel values; in such cases the `pmTable` field points to a dummy color table.

Note

The pixel map for a window's color graphics port always consists of the pixel depth, color table, and boundary rectangle of the main screen, even if the window is created on or moved to an entirely different screen. ◆

Pixel Patterns

Color QuickDraw supplements the black-and-white patterns of basic QuickDraw with pixel patterns, which can use colors at any pixel depth and can be of any width and height that's a power of 2. A **pixel pattern** defines a repeating design (such as stripes of different colors) or a color otherwise unavailable on indexed devices. For example, if your application draws to an indexed device that supports 4 bits per pixel, your application has 16 colors available if it simply sets the foreground color and draws. However, if your application uses the `MakeRGBPat` procedure to create patterns that use these 16 colors in various combinations, and then draws using that pattern, your application can effectively have as many as 125 approximated colors at its disposal. For example, you can specify a purple color to `MakeRGBPat`, which creates a pattern that mixes blue and red pixels.

As with bit patterns (described in the chapter "QuickDraw Drawing"), your application can use pixel patterns to draw lines and shapes on the screen. In a color graphics port, the graphics pen has a pixel pattern specified in the `pnPixPat` field of the `CGrafPort` record. This pixel pattern acts like the ink in the pen; the pixels in the pattern interact with the pixels in the pixel map 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 (described in the chapter "QuickDraw Drawing") to draw shapes, these procedures draw the shape with the pattern specified in the `pnPixPat` field. Initially, every graphics pen is assigned an all-black pattern, but you can use the `PenPixPat` procedure to assign a different pixel pattern to the graphics pen.

You can use the `FillCRect`, `FillCRoundRect`, `FillCArc`, `FillCPoly`, and `FillCRgn` procedures (described later in this chapter) to draw shapes with a pixel pattern other than the one specified in the `pnPixPat` field. When your application uses one of these procedures, the procedure stores the pattern your application specifies in the `fillPixPat` field of the `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 (for example, by the `EraseRect`, `EraseRoundRect`, `EraseArc`, `ErasePoly`, and `EraseRgn` procedures, described in the chapter "QuickDraw Drawing") and when pixels are scrolled out of an area by the `ScrollRect` procedure, described in the chapter "Basic QuickDraw." Every color graphics port stores a background pixel pattern in the `bkPixPat` field of its `CGrafPort` record. Initially, every graphics port is assigned an all-white background pattern, but you can use the `BackPixPat` procedure to assign a different pixel pattern.

You can create your own pixel patterns in your program code, but it's usually simpler and more convenient to store them in resources of type `'ppat'`.

Each pixel map has its own color table; therefore, pixel patterns can consist of any number of colors, and they don't usually require the graphics port's foreground and background colors to have particular values.

Note
Color QuickDraw also supports bit patterns. When used in a `CGrafPort` record, such patterns are limited to 8-by-8 bit dimensions and are always drawn using the values in the `fgColor` and `bkColor` fields of the `CGrafPort` record. ◆

Color QuickDraw's Translation of RGB Colors to Pixel Values

When using Color QuickDraw, your application refers to a color only through the three 16-bit fields of a 48-bit `RGBColor` record; you use these fields to specify the red, green, and blue components of your desired color. When your application draws into a pixel map, Color QuickDraw and the Color Manager translate your `RGBColor` records into pixel values; these pixel values are sent to your users' graphics devices, which display the pixels accordingly.

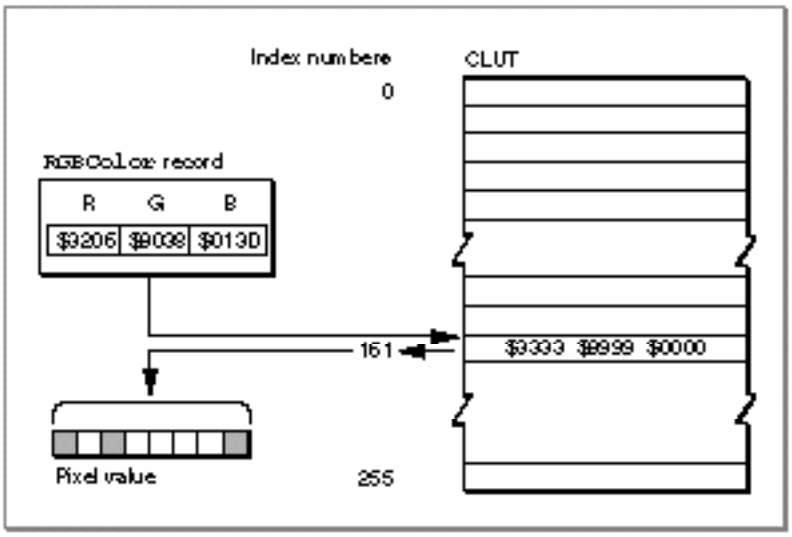
Your application never needs to handle pixel values. However, to clarify the relation between your application's 48-bit `RGBColor` records and the pixels that are actually displayed, this section presents some examples of how Color QuickDraw derives pixel values from your `RGBColor` records.

Indexed devices were introduced to support—with minimal memory requirements—the color capabilities of the Macintosh II computer. The pixel value for any color on an indexed device is represented by a single byte. Each byte contains an index number that specifies one of 256 colors available on the device's CLUT. This index number is the pixel value for the pixel. (Some indexed devices support 1-bit, 2-bit, or 4-bit pixel values, resulting in tables containing 2, 4, or 16 colors, respectively, as shown in Plate 1 in the front of this book.)

To obtain an 8-bit pixel value from the 48-bit `RGBColor` record specified by your application, Color QuickDraw calls on the Color Manager to determine the closest RGB color stored in the CLUT on the current device. The index number to that color is then stored in the 8-bit pixel.

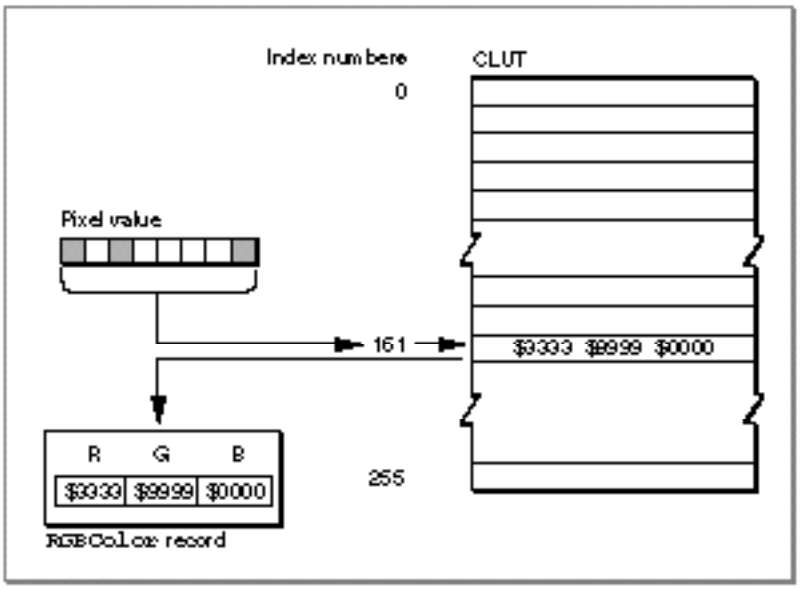
For example, the `RGBColor` record for a medium green pixel is represented on the left side of Figure 4-3. An application might create such a record and pass it to the `RGBForeColor` procedure, which sets the foreground color for drawing. In system software's standard 8-bit color lookup table (which is defined in a `'clut'` resource with the resource ID of 8), the closest color to that medium green is stored as table entry 161. When the next pixel is drawn, this index number is stored in the pixel image as the pixel value.

Figure 4-3 Translating a 48-bit `RGBColor` record to an 8-bit pixel value on an indexed device



The application might later use the `GetCPixel` procedure to determine the color of a particular pixel. As shown in Figure 4-4, the Color Manager uses the index number stored as the pixel value to find the 48-bit `RGBColor` record stored in the CLUT for that pixel's color—which, as with the medium green in this example, is not necessarily the exact color first specified by the application. The difference, however, is imperceptible.

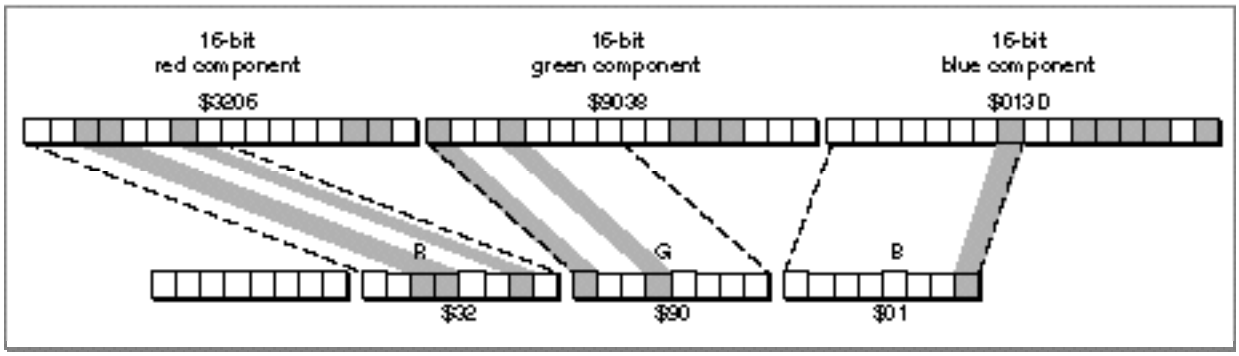
Figure 4-4 Translating an 8-bit pixel value on an indexed device to a 48-bit `RGBColor` record



Direct devices support 32-bit and 16-bit pixel values. Direct devices do not use tables to store and look up colors, nor do their pixel values consist of index numbers. For each pixel on a direct device, Color QuickDraw instead derives the pixel value by concatenating the values of the `red`, `green`, and `blue` fields of an `RGBColor` record.

As shown in Figure 4-5, Color QuickDraw converts a 48-bit `RGBColor` record into a 32-bit pixel value by storing the most significant 8 bits of each 16-bit field of the `RGBColor` record into the lower 3 bytes of the pixel value, leaving 8 unused bits in the high byte of the pixel value.

Figure 4-5 Translating a 48-bit `RGBColor` record to a 32-bit pixel value on a direct device



Color QuickDraw converts a 48-bit `RGBColor` record into a 16-bit pixel value by storing the most significant 5 bits of each 16-bit field of the `RGBColor` record into the lower 15 bits of the pixel value, leaving an unused high bit, as shown in Figure 4-6.

Figure 4-6 Translating a 48-bit `RGBColor` record to a 16-bit pixel value on a direct device

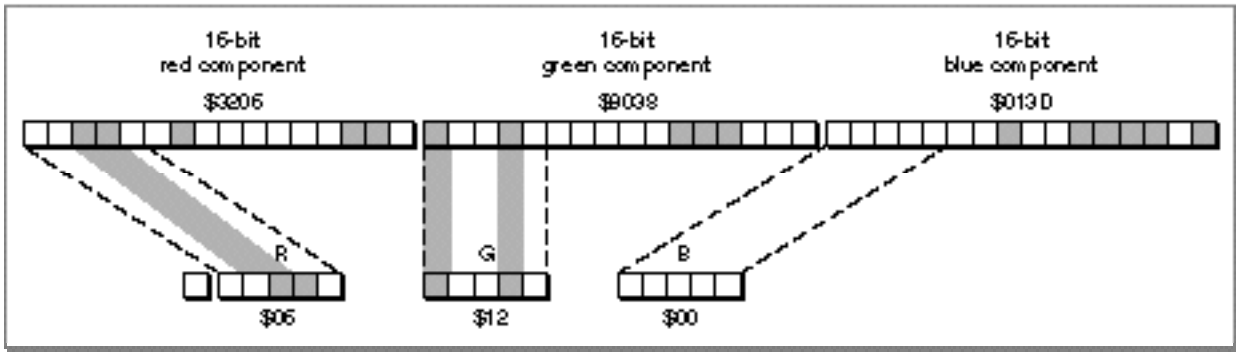


Figure 4-7 shows how Color QuickDraw expands a 32-bit pixel value to a 48-bit `RGBColor` record by dropping the unused high byte of the pixel value and doubling each of its 8-bit components. Note that the resulting 48-bit value differs in the least significant 8 bits of each component from the original `RGBColor` record in Figure 4-5.

Figure 4-7 Translating a 32-bit pixel value to a 48-bit `RGBColor` record

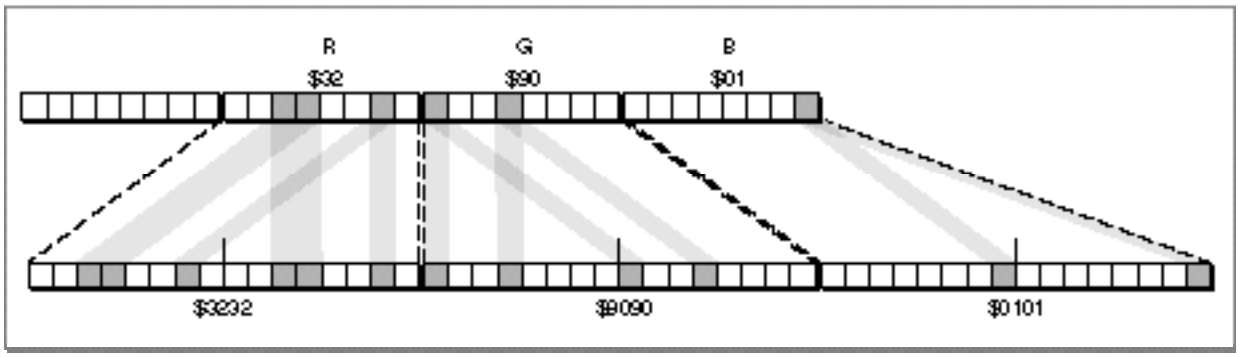
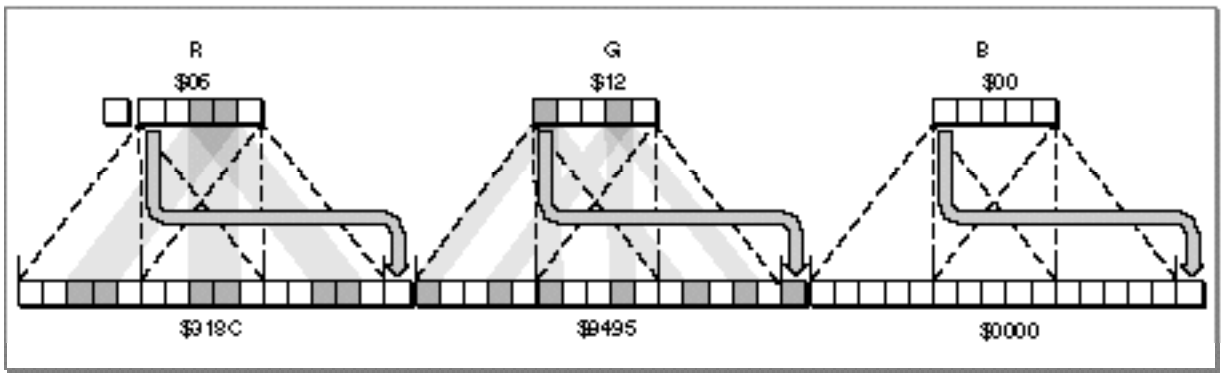


Figure 4-8 shows how Color QuickDraw expands a 16-bit pixel value to a 48-bit `RGBColor` record by dropping the unused high bit of the pixel value and inserting three copies of each 5-bit component and a copy of the most significant bit into each 16-bit field of the `RGBColor` record. Note that the result differs (in the least significant 11 bits of each component) from the original 48-bit value in Figure 4-5. The difference, however, is imperceptible.

Figure 4-8 Translating a 16-bit pixel value to a 48-bit `RGBColor` record



Colors on Grayscale Screens

When Color QuickDraw displays a color on a grayscale screen, it computes the **luminance,** or intensity of light, of the desired color and uses that value to determine the appropriate gray value to draw. A grayscale graphics device can be a color graphics device that the user sets to grayscale by using the Monitors control panel; for such a graphics device, Color QuickDraw places an evenly spaced set of grays, forming a linear ramp from white to black, in the graphics device's CLUT. (When a user uses the Monitors control panel to set a 16-bit or 32-bit direct device to use 2, 4, 16, or 256 colors as a grayscale or color device, the direct device creates a CLUT and operates like an indexed device.)

By using the `GetCTable` function, described on page 4-92, your application can obtain the default color tables for various graphics devices, including grayscale devices.

Using Color QuickDraw

To use Color QuickDraw, you generally

- initialize QuickDraw
- create a color window into which your application can draw
- create `RGBColor` records to define your own foreground and background colors
- create pixel pattern (`'ppat'`) resources to define your own colored patterns
- use these colors and pixel patterns for drawing with the graphics pen, for filling as the background pattern, and for filling into shapes
- use the basic QuickDraw routines previously described in this book to perform all other onscreen graphics port manipulations and calculations

This section gives an overview of routines that your application typically calls while using Color QuickDraw. Before calling these routines, however, your application should test for the existence of Color QuickDraw by using the `Gestalt` function with the `gestaltQuickDrawVersion` selector. The `Gestalt` function returns a 4-byte value in its `response` parameter; the low-order word contains QuickDraw version data. In that low-order word, the high-order byte gives the major revision number and the low-order byte gives the minor revision number. If the value returned in the `response` parameter is equal to the value of the constant `gestalt32BitQD13`, then the system supports the System 7 version of Color QuickDraw. Listed here are the various constants, and the values they represent, that indicate earlier versions of Color QuickDraw.

```
CONST
   gestalt8BitQD     = $100;  {8-bit Color QD}
   gestalt32BitQD    = $200;  {32-bit Color QD}
   gestalt32BitQD11  = $210;  {32-bit Color QDv1.1}
   gestalt32BitQD12  = $220;  {32-bit Color QDv1.2}
   gestalt32BitQD13  = $230;  {System 7: 32-bit Color QDv1.3}
```

Your application can also use the `Gestalt` function with the selector `gestaltQuickDrawFeatures` to determine whether the user's system supports various Color QuickDraw features. If the bits indicated by the following constants are set in the `response` parameter, then the features are available:

```
CONST
   gestaltHasColor         = 0;  {Color QuickDraw is present}
   gestaltHasDeepGWorlds   = 1;  {GWorlds deeper than 1 bit}
   gestaltHasDirectPixMaps = 2;  {PixMaps can be direct--16 or }
                                 { 32 bit}
   gestaltHasGrayishTextOr = 3;  {supports text mode }
                                 { grayishTextOr}
```

When testing for the existence of Color QuickDraw, your application should test the response to the `gestaltQuickDrawVersion` selector (rather than test for the result `gestaltHasColor`, which is unreliable, from the `gestaltQuickDrawFeatures` selector). The support for offscreen graphics worlds indicated by the `gestaltHasDeepGWorlds` response to the `gestaltQuickDrawVersion` selector is described in the chapter "Offscreen Graphics Worlds." The support for the text mode indicated by the `gestaltHasGrayishTextOr` response is described in the chapter "QuickDraw Text" in *Inside Macintosh: Text*. For more information about the `Gestalt` function, see the chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities*.

Initializing Color QuickDraw

To initialize Color QuickDraw, use the `InitGraf` procedure, described in the chapter "Basic QuickDraw." Besides initializing basic QuickDraw, this procedure initializes Color QuickDraw on computers that support it.

In addition to `InitGraf`, all other basic QuickDraw routines work with Color QuickDraw. For example, you can use the `GetPort` procedure to save the current color graphics port, and you can use the `CopyBits` procedure to copy an image between two different color graphics ports. See the chapters "Basic QuickDraw" and "QuickDraw Drawing" for descriptions of additional routines that you can use with Color QuickDraw.

Creating Color Graphics Ports

All graphics operations are performed in graphics ports. Before a color graphics port can be used, it must be allocated with the `OpenCPort` procedure and initialized with the `InitCPort` procedure. Normally, your application does not call these procedures directly. Instead, your application creates a color graphics port by using the `GetNewCWindow` or `NewCWindow` function (described in the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*) or the `NewGWorld` function (described in the chapter "Offscreen Graphics Worlds" in this book). These functions automatically call `OpenCPort`, which in turn calls `InitCPort`.

Listing 4-1 shows a simplified application-defined procedure called `DoNew` that uses the Window Manager function `GetNewCWindow` to create a color graphics port.

Listing 4-1 Using the Window Manager to create a color graphics port

```
PROCEDURE DoNew (VAR window: WindowPtr);
VAR
   windStorage:   Ptr;  {memory for window record}
BEGIN
   window := NIL;
   {allocate memory for window record from previously allocated block}
   windStorage := MyPtrAllocationProc;
   IF windStorage <> NIL THEN {memory allocation succeeded}
   BEGIN
      IF gColorQDAvailable THEN {use Gestalt to determine color availability}
         window := GetNewCWindow(rDocWindow, windStorage, WindowPtr(-1))
      ELSE        {create a basic graphics port for a black-and-white screen}
         window := GetNewWindow(rDocWindow, windStorage, WindowPtr(-1));
   END;
   IF (window <> NIL) THEN
       SetPort(window);
END;
```

You can use `GetNewCWindow` to create color graphics ports whether or not a color monitor is currently installed. So that most of your window-handling code can handle color windows and black-and-white windows identically, `GetNewCWindow` returns a pointer of type `WindowPtr` (not of type `CWindowPtr`).

A window pointer points to a window record (`WindowRecord`), which contains a `GrafPort` record. If you need to check the fields of the color graphics port associated with a window, you can coerce the pointer to the `GrafPort` record into a pointer to a `CGrafPort` record.

You can allow `GetNewCWindow` to allocate the memory for your window record and its associated basic graphics port. You can maintain more control over memory use, however, by allocating the memory yourself from a block allocated for such purposes during your own initialization routine, and then passing the pointer to `GetNewWindow`, as shown in Listing 4-1.

To dispose of a color graphics port when you are finished using a color window, you normally use the `DisposeWindow` procedure (if you let the Window Manager allocate memory for the window) or the `CloseWindow` procedure (if you allocated memory for the window). If you use the `CloseWindow` procedure, you also dispose of the window record containing the graphics port by calling the Memory Manager procedure `DisposePtr`. You use the `DisposeGWorld` procedure when you are finished with a color graphics port for an offscreen graphics world.

Drawing With Different Foreground Colors

You can set the foreground and background colors using either Color QuickDraw or Palette Manager routines. If your application uses the Palette Manager, it should set the foreground and background colors with the `PmForeColor` and `PmBackColor` routines, as described in the chapter "Palette Manager" in *Inside Macintosh: Advanced Color Imaging*. Otherwise, your application can use the `RGBForeColor` procedure to set the foreground color, and it can use the `RGBBackColor` procedure to set the background color. Both of these Color QuickDraw procedures also operate for basic graphics ports created in System 7. (To set the foreground and background colors for basic graphics ports on older versions of system software, use the `ForeColor` and `BackColor` procedures described in the chapter "QuickDraw Drawing.")

The `RGBForeColor` procedure lets you set the foreground color to the best color available on the current graphics device. This changes the color of the "ink" used for drawing. All of the line-drawing, framing, and painting routines described in the chapter "QuickDraw Drawing" (such as `LineTo`, `FrameRect`, and `PaintPoly`) draw with the foreground color that you specify with `RGBForeColor`.

Note

Because a pixel pattern already contains color, Color QuickDraw ignores the foreground and background colors when your application draws with a pixel pattern. As described in "Drawing With Pixel Patterns" beginning on page 4-23, you can draw with a pixel pattern by using the `PenPixPat` procedure to assign a pixel pattern to the graphics pen, by using the `BackPixPat` procedure to assign a pixel pattern as the background pattern for the current color graphics port, and by using the `FillCRect`, `FillCOval`, `FillCRoundRect`, `FillCArc`, `FillCRgn`, and `FillCPoly` procedures to fill shapes with a pixel pattern. ◆

To specify a foreground color, create an RGBColor record. Listing 4-2 defines two RGBColor records. The first is declared as myDarkBlue, and it's defined with a medium-intensive blue component and with zero-intensity red and green components. The second is declared as myMediumGreen, and it's defined with an intensive green component, a mildly intensive red component, and a very slight blue component.

Listing 4-2 Changing the foreground color

```
PROCEDURE MyPaintAndFillColorRects;
VAR
   firstRect, secondRect:  Rect;
   myDarkBlue:             RGBColor;
   myMediumGreen:          RGBColor;
BEGIN
      {create dark blue color}
   myDarkBlue.red := $0000;
   myDarkBlue.green := $0000;
   myDarkBlue.blue := $9999;
      {create medium green color}
   myMediumGreen.red := $3206;
   myMediumGreen.green := $9038;
   myMediumGreen.blue := $013D;
   RGBForeColor(myDarkBlue);  {draw with dark blue pen}
   PenMode(patCopy);
   SetRect(firstRect, 20, 20, 70, 70);
   PaintRect(firstRect);         {paint a dark blue rectangle}
   RGBForeColor(myMediumGreen);  {draw with a medium green pen}
   SetRect(secondRect, 90, 20, 140, 70);
   FillRect(secondRect, ltGray); {paint a medium green rectangle}
END;
```

In Listing 4-2, the RGBColor record myDarkBlue is supplied to the RGBForeColor procedure. The RGBForeColor procedure supplies the rgbFgColor field of the CGrafPort record with this RGBColor record, and it places the closest-matching available color in the fgColor field; the color in the fgColor field is the color actually used as the foreground color.

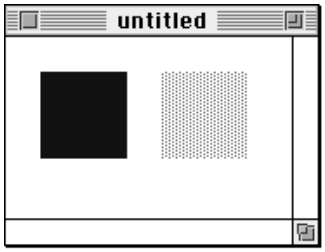
After using SetRect to create a rectangle, Listing 4-2 calls PaintRect to paint the rectangle. By default, the foreground color is black; by changing the foreground color to dark blue, every pixel that would normally be painted in black is instead painted in dark blue.

Listing 4-2 then changes the foreground color again to the medium green specified in the RGBColor record myMediumGreen. After creating another rectangle, this listing calls FillRect to fill the rectangle with the bit pattern specified by the global variable ltGray. As explained in the chapter "QuickDraw Drawing," this bit pattern consists of

widely spaced black pixels that create the effect of gray on black-and-white screens. However, by changing the foreground color, every pixel in the pattern that would normally be painted black is instead drawn in medium green.

The effects of Listing 4-2 are illustrated in the grayscale screen capture shown in Figure 4-9.

Figure 4-9 Drawing with two different foreground colors (on a grayscale screen)



If you wish to draw with a color other than the foreground color, you can use the `PenPixPat` procedure to give the graphics pen a colored pixel pattern that you define, and you can use the `FillCRect`, `FillCRoundRect`, `FillCOval`, `FillCArc`, `FillCPoly`, and `FillCRgn` procedures to fill shapes with colored patterns. The use of these procedures is illustrated in the next section.

Drawing With Pixel Patterns

Using pixel pattern resources, you can create multicolored patterns for the pen pattern, for the background pattern, and for fill patterns.

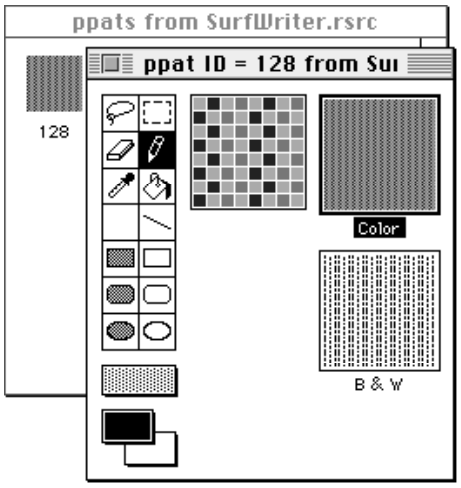
To set the pixel pattern to be used by the graphics pen in the current color graphics port, you use the `PenPixPat` procedure. To assign a pixel pattern as the background pattern, you use the `BackPixPat` procedure; this causes the `ScrollRect` procedure and the shape-erasing procedures (for example, `EraseRect`) to fill the background with your pixel pattern. To fill shapes with a pixel pattern, you use the `FillCRect`, `FillCRoundRect`, `FillCOval`, `FillCArc`, `FillCPoly`, and `FillCRgn` procedures.

Note
Because a pixel pattern already contains color, Color QuickDraw ignores the foreground and background colors when your application uses these routines to draw with a pixel pattern. Color QuickDraw also ignores the pen mode by drawing the pixel pattern directly onto the pixel image. ◆

When you use the `PenPat` or `BackPat` procedure in a color graphics port, Color QuickDraw constructs a pixel pattern equivalent to the bit pattern you specify to `PenPat` or `BackPat`. The pen pattern or background pattern you thereby specify always uses the graphics port's current foreground and background colors. The `PenPat` and `BackPat` procedures are described in the chapter "QuickDraw Drawing."

A pixel pattern resource is a resource of type `'ppat'`. You typically use a high-level tool such as the ResEdit application, available through APDA, to create `'ppat'` resources. Figure 4-10 illustrates a ResEdit window displaying an application's `'ppat'` resource with resource ID 128.

Figure 4-10 Using ResEdit to create a pixel pattern resource



As shown in this figure, you should also define an analogous, black-and-white bit pattern (described in the chapter "QuickDraw Drawing") to be used when this pattern is drawn into a basic graphics port. This bit pattern is stored within the pixel pattern resource.

After using ResEdit to define a pixel pattern, you can then use the DeRez decompiler to convert your `'ppat'` resources into Rez input when necessary. (The DeRez resource decompiler and the Rez resource compiler are part of Macintosh Programmer's Workshop [MPW], which is available through APDA.) Listing 4-3 shows the Rez input created from the `'ppat'` resource created in Figure 4-10.

Listing 4-3 Rez input for a pixel pattern resource

```
resource 'ppat' (128) {
   $"0001 0000 001C 0000 004E 0000 0000 FFFF"
   $"0000 0000 8292 1082 9210 8292 0000 0000"
   $"8002 0000 0000 0008 0008 0000 0000 0000"
   $"0000 0048 0000 0048 0000 0000 0002 0001"
   $"0002 0000 0000 0000 005E 0000 0000 1212"
   $"4848 1212 4848 1212 4848 1212 4848 0000"
```

```
   $"0000 0000 0002 0000 AAAA AAAA AAAA 0001"
   $"2222 2222 2222 0002 7777 7777 7777"
};
```

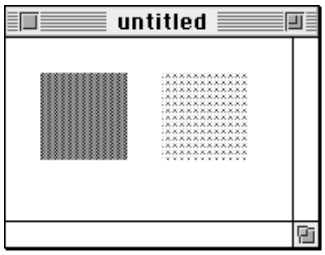
To retrieve the pixel pattern stored in a `'ppat'` resource, you can use the `GetPixPat` function. Listing 4-4 uses `GetPixPat` to retrieve the `'ppat'` resource created in Listing 4-3. To assign this pixel pattern to the graphics pen, Listing 4-4 uses the `PenPixPat` procedure.

Listing 4-4 Using pixel patterns to paint and fill

```
PROCEDURE MyPaintPixelPatternRects;
VAR
   firstRect, secondRect:        Rect;
   myPenPattern, myFillPattern:  PixPatHandle;
BEGIN
   myPenPattern := GetPixPat(128);  {get a pixel pattern}
   PenPixPat(myPenPattern);      {assign the pattern to the pen}
   SetRect(firstRect, 20, 20, 70, 70);
   PaintRect(firstRect);      {paint with the pen's pixel pattern}
   DisposePixPat(myPenPattern);  {dispose of the pixel pattern}
   myFillPattern := GetPixPat(129);    {get another pixel pattern}
   SetRect(secondRect, 90, 20, 140, 70);
   FillCRect(secondRect, myFillPattern);  {fill with this pattern}
   DisposePixPat(myFillPattern); {dispose of the pixel pattern}
END;
```

Listing 4-4 uses the `PaintRect` procedure to draw a rectangle. The rectangle on the left side of Figure 4-11 illustrates the effect of painting a rectangle with the previously defined pen pattern.

Figure 4-11 Painting and filling rectangles with pixel patterns



The rectangle on the right side of Figure 4-11 illustrates the effect of using the `FillCRect` procedure to fill a rectangle with another previously defined pen pattern. The `GetPixPat` function is used to retrieve the pixel pattern defined in the `'ppat'` resource with resource ID 129. This pixel pattern is then specified to the `FillCRect` procedure.

Copying Pixels Between Color Graphics Ports

As explained in the chapter "QuickDraw Drawing," QuickDraw has three primary image-processing routines.

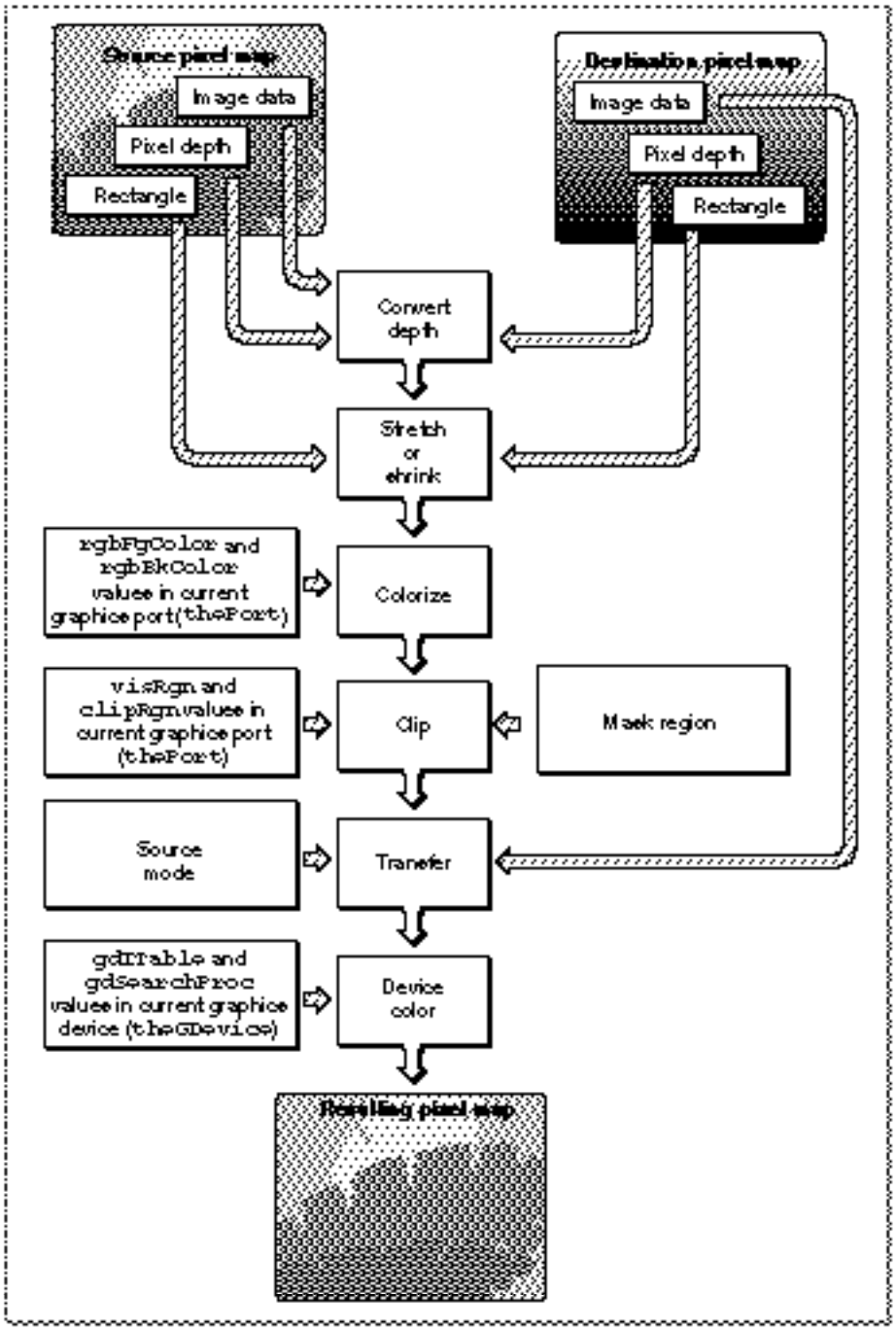
- The `CopyBits` procedure copies a pixel map or bitmap image to another graphics port, with facilities for resizing the image, modifying the image with transfer modes, and clipping the image to a region.
- The `CopyMask` procedure copies a pixel map or bitmap image to another graphics port, with facilities for resizing the image and for altering the image by passing it through a mask—which for Color QuickDraw may be another pixel map whose pixels indicate proportionate weights of the colors for the source and destination pixels.
- The `CopyDeepMask` procedure combines the effects of `CopyBits` and `CopyMask`: you can resize an image, clip it to a region, specify a transfer mode, and use another pixel map as a mask when transferring it to another graphics port.

In basic QuickDraw, `CopyBits`, `CopyMask`, and `CopyDeepMask` copy bit images between two basic graphics ports. In Color QuickDraw, you can also use these procedures to copy pixel images between two color graphics ports. Detailed routine descriptions for these procedures appear in the chapter "QuickDraw Drawing." This section provides an overview of how to use the extra capabilities that Color QuickDraw provides for these procedures.

When using `CopyBits`, `CopyMask`, and `CopyDeepMask` to copy images between color graphics ports, you must coerce each port's `CGrafPtr` data type to a `GrafPtr` data type, 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`, in the `srcBits` parameter you could specify `GrafPtr(MyColorPort)^.portBits`. 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 these routines that you have passed it a handle to a pixel map rather than a bitmap.

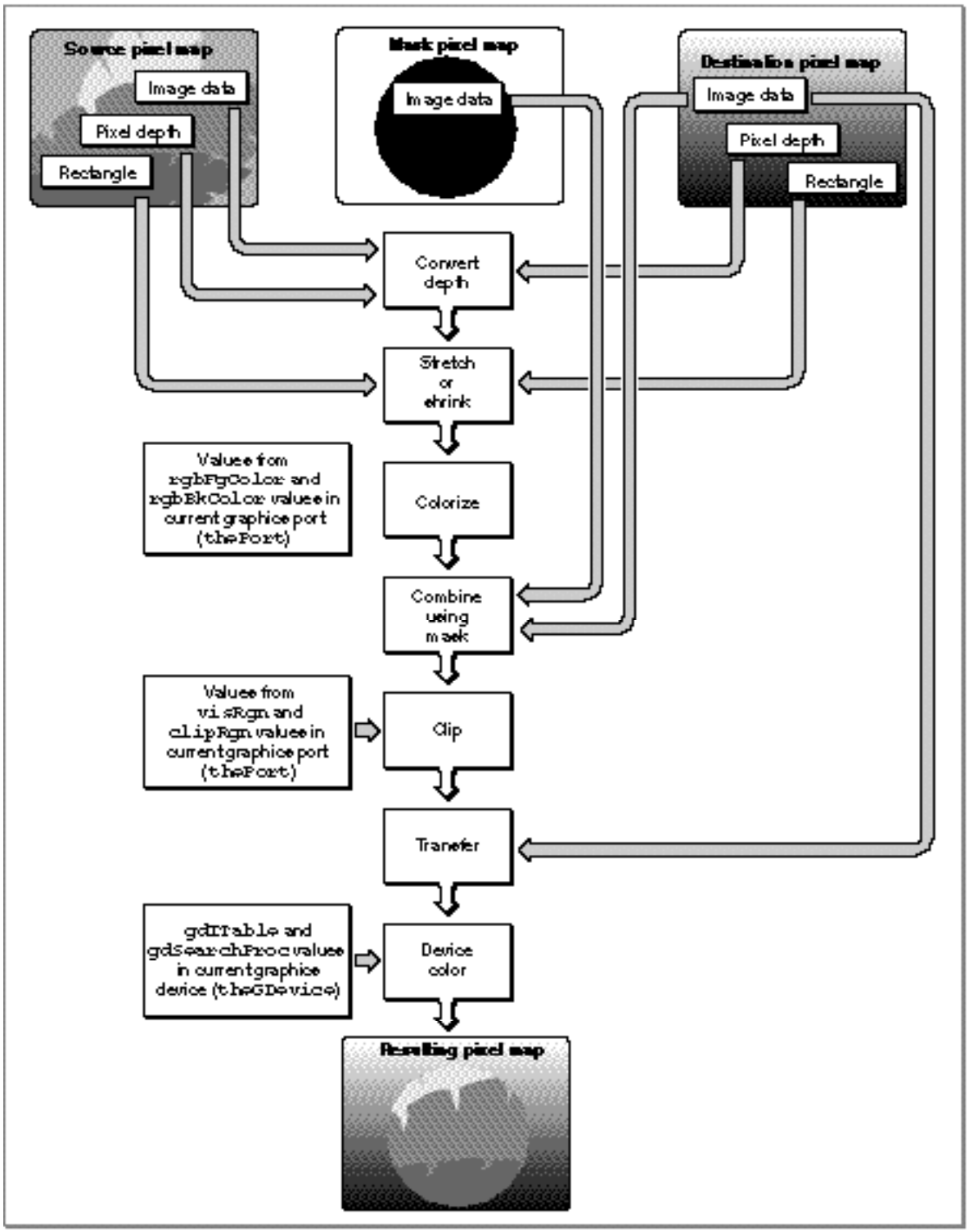
Color QuickDraw's processing sequence of the `CopyBits` procedure is illustrated in Figure 4-12. Listing 6-1 in the chapter "Offscreen Graphics Worlds" illustrates how to use `CopyBits` to transfer an image prepared in an offscreen graphics world to an onscreen color graphics port.

Figure 4-12 Copying pixel images with the `CopyBits` procedure



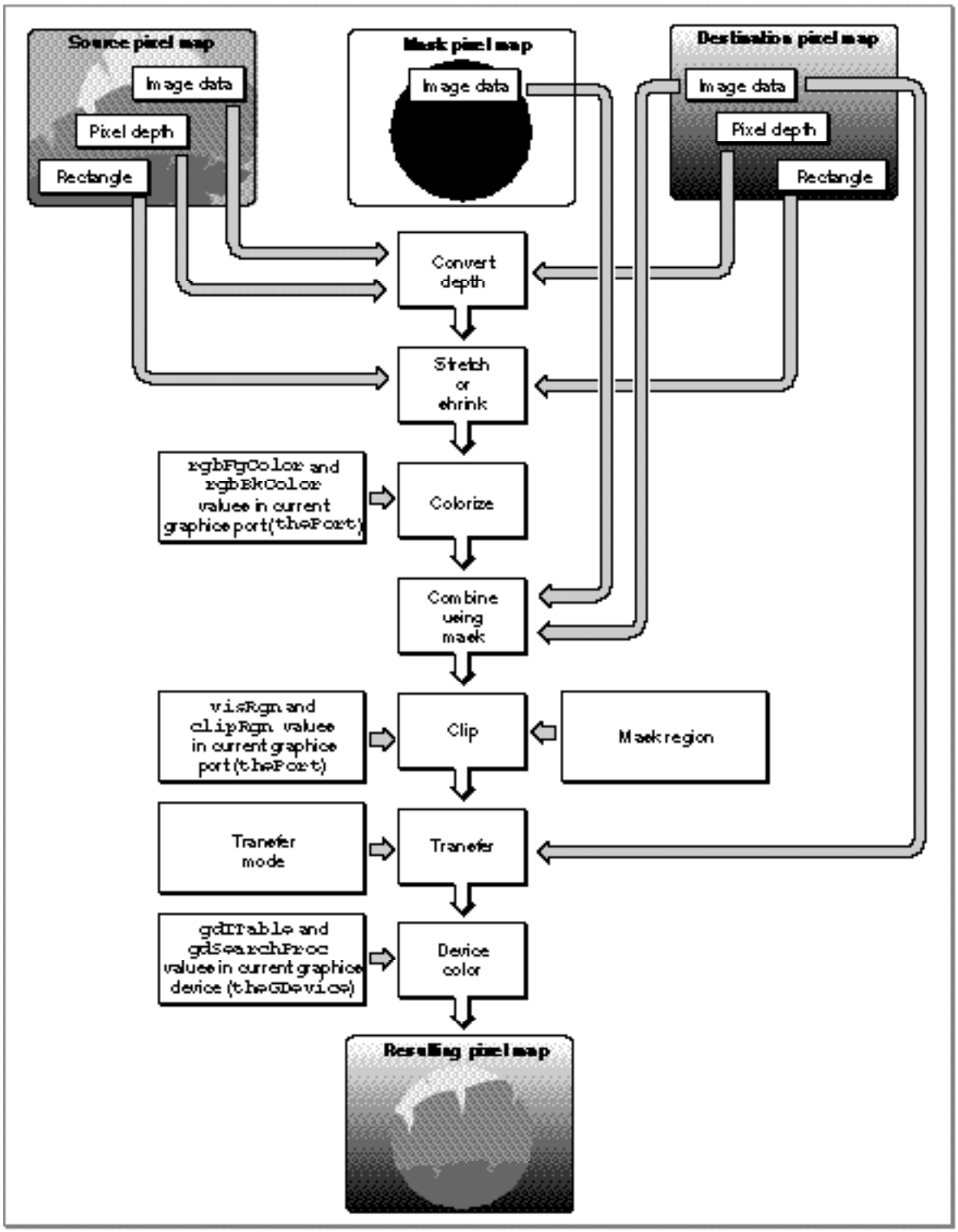
With the `CopyMask` procedure, you can supply a pixel map to act as a copying mask. The values of pixels in the mask act as weights that proportionally select between source and destination pixel values. The process is shown in Figure 4-13, and an example of the effect can be seen in Plate 3 at the front of this book. Listing 6-2 in the chapter "Offscreen Graphics Worlds" illustrates how to use `CopyMask` to mask and copy an image prepared in an offscreen graphics world to an onscreen color graphics port.

Figure 4-13 Copying pixel images with the `CopyMask` procedure



The `CopyDeepMask` procedure combines the capabilities of the `CopyBits` and `CopyMask` procedures. With `CopyDeepMask` you can specify a pixel map mask, a transfer mode, and a mask region, as shown in Figure 4-14.

Figure 4-14 Copying pixel images with the `CopyDeepMask` procedure



On indexed devices, pixel images are always copied using the color table of the source `PixMap` record for source color information, and using the color table of the *current* `GDevice` *record* for destination color information. The color table attached to the destination `PixMap` record is ignored. As explained in the chapter "Offscreen Graphics Worlds," if you need to copy to an offscreen `PixMap` record with characteristics differing from those of the current graphics device, you should create an appropriate offscreen `GDevice` record and set it as the current graphics device before the copy operation.

When the `PixMap` record for the mask is 1 bit deep, it has the same effect as a bitmap mask: a black bit in the mask means that the destination pixel is to take the color of the source pixel; a white bit in the mask means that the destination pixel is to retain its current color. When masks have `PixMap` records with greater pixel depths than 1, Color QuickDraw takes a weighted average between the colors of the source and destination `PixMap` records. Within each pixel, the calculation is done in RGB color, on a color component basis. A gray `PixMap` record mask, for example, works like blend mode in a `CopyBits` procedure. A red mask (that is, one with high values for the red components of all pixels) filters out red values coming from the source pixel image.

Boolean Transfer Modes With Color Pixels

As described in the chapter "QuickDraw Drawing," QuickDraw offers two types of Boolean transfer modes: pattern modes for drawing lines and shapes, and source modes for copying images or drawing text. In basic graphics ports and in color graphics ports with 1-bit pixel maps, these modes describe the interaction between the bits your application draws and the bits that are already in the destination bitmap or 1-bit pixel map. These interactions involve turning the bits on or off—that is, making the pixels black or white.

The Boolean operations on bitmaps and 1-bit pixel maps are described in the chapter "QuickDraw Drawing." When you draw or copy images to and from bitmaps or 1-bit pixel maps, Color QuickDraw behaves in the manner described in that chapter.

When you use pattern modes in pixel maps with depths greater than 1 bit, Color QuickDraw uses the foreground color and background color when transferring bit patterns; for example, the `patCopy` mode applies the foreground color to every destination pixel that corresponds to a black pixel in a bit pattern, and it applies the background color to every destination pixel that corresponds to a white pixel in a bit pattern. See the description of the `PenMode` procedure in the chapter "QuickDraw Drawing" for a list that summarizes how the foreground and background colors are applied with pattern modes.

When you use the `CopyBits`, `CopyMask`, and `CopyDeepMask` procedures to transfer images between pixel maps with depths greater than 1 bit, Color QuickDraw performs the Boolean transfer operations in the manner summarized in Table 4-1. In general, with pixel images you will probably want to use the `srcCopy` mode or one of the arithmetic transfer modes described in "Arithmetic Transfer Modes" beginning on page 4-38.

Table 4-1 Boolean source modes with colored pixels

Source mode	Action on destination pixel		
	If source pixel is black	If source pixel is white	If source pixel is any other color
`srcCopy`	Apply foreground color	Apply background color	Apply weighted portions of foreground and background colors
`notSrcCopy`	Apply background color	Apply foreground color	Apply weighted portions of background and foreground colors
`srcOr`	Apply foreground color	Leave alone	Apply weighted portions of foreground color
`notSrcOr`	Leave alone	Apply foreground color	Apply weighted portions of foreground color
`srcXor`	Invert (undefined for colored destination pixel)	Leave alone	Leave alone
`notSrcXor`	Leave alone	Invert (undefined for colored destination pixel)	Leave alone
`srcBic`	Apply background color	Leave alone	Apply weighted portions of background color
`notSrcBic`	Leave alone	Apply background color	Apply weighted portions of background color

Note
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. ◆

When you use the `srcCopy` mode to transfer a pixel into a pixel map, Color QuickDraw determines how close the color of that pixel is to black, and then assigns this relative amount of foreground color to the destination pixel. Color QuickDraw also determines how close the color of that pixel is to white, and assigns this relative amount of background color to the destination pixel.

To accomplish this, Color QuickDraw first multiplies the relative intensity of each red, green, and blue component of the source pixel by the corresponding value of the red, green, or blue component of the foreground color. It then multiplies the relative intensity of each red, green, and blue component of the source pixel by the corresponding value of the red, green, or blue component of the background color. For each component, Color QuickDraw adds the results and then assigns the new result as the value for the destination pixel's corresponding component.

For example, the pixel in an image might be all red: that is, its red component has a pixel value of $FFFF, and its green and blue components each have pixel values of $0000. The current foreground color might be black (that is, with pixel values of $0000, $0000, $0000 for its components) and its background color might be all white (that is, with pixel values of $FFFF, $FFFF, $FFFF). When that image is copied using the `CopyBits` procedure and the `srcCopy` source mode, `CopyBits` determines that the red component of the source pixel has 100 percent intensity; multiplying this by the intensity of the red component ($0000) of the foreground color produces a value of $0000, and multiplying this by the intensity of the red component ($FFFF) of the background color produces a value of $FFFF. Adding the results of these two operations produces a pixel value of $FFFF for the red component of the destination pixel. Performing similar operations on the green and blue components of the source pixel produces green and blue pixel values of $0000 for the destination pixel. In this way, `CopyBits` renders the source's all-red pixel as an all-red pixel in the destination pixel map. A source pixel with only 50 percent intensity for its red component and no intensity for its blue and green components would similarly be drawn as a medium red pixel in the destination pixel map.

Color QuickDraw produces similarly weighted destination colors when you use the other Boolean source modes. When you use the `srcBic` mode to transfer a colored source pixel into a pixel map, for example, `CopyBits` determines how close the color of that pixel is to black, and then assigns a relative amount of background color to the destination pixel. For this mode, `CopyBits` does not determine how close the color of the source pixel is to white.

Because Color QuickDraw uses the foreground and background colors instead of black and white when performing its Boolean source operations, the following effects are produced:

- The `notSrcCopy` mode reverses the foreground and background colors.
- Drawing into a white background with a black foreground always reproduces the source image, regardless of the pixel depth.
- Drawing is faster if the foreground color is black when you use the `srcOr` and `notSrcOr` modes.
- If the background color is white when you use the `srcBic` mode, the black portions of the source are erased, resulting in white in the destination pixel map.

As you can see, applying a foreground color other than black or a background color other than white to the pixel can produce an unexpected result. For consistent results, set the foreground color to black and the background color to white before using `CopyBits`, `CopyMask`, or `CopyDeepMask`.

However, by using the `RGBForeColor` and `RGBBackColor` procedures to set the foreground and background colors to something other than black and white before using `CopyBits`, `CopyMask`, or `CopyDeepMask`, you can achieve some interesting coloration effects. Plate 2 at the front of this book shows how setting the foreground color to red and the background color to blue and then using the `CopyBits` procedure turns a grayscale image into shades of red and blue. Listing 4-5 shows the code that produced these two pixel maps.

Listing 4-5 Using `CopyBits` to produce coloration effects

```
PROCEDURE MyColorRamp;
VAR
   origPort:            CGrafPtr;
   origDevice:          GDHandle;
   myErr:               QDErr;
   myOffScreenWorld:    GWorldPtr;
   TheColor:            RGBColor;
   i:                   Integer;
   offPixMapHandle:     PixMapHandle;
   good:                Boolean;
   myRect:              Rect;
BEGIN
   GetGWorld(origPort, origDevice);    {save onscreen graphics port}
                                {create offscreen graphics world}
   myErr := NewGWorld(myOffScreenWorld,
                      0, origPort^.portRect, NIL, NIL, []);
   IF (myOffScreenWorld = NIL) OR (myErr <> noErr) THEN
      ; {handle errors here}
   SetGWorld(myOffScreenWorld, NIL); {set current graphics port to offscreen}
   offPixMapHandle := GetGWorldPixMap(myOffScreenWorld);
   good := LockPixels(offPixMapHandle);   {lock offscreen pixel map}
   IF NOT good THEN
      ; {handle errors here}
   EraseRect(myOffScreenWorld^.portRect); {initialize offscreen pixel map}
   FOR i := 0 TO 9 DO
   BEGIN                                  {create gray ramp}
      theColor.red := i * 7168;
      theColor.green := i * 7168;
      theColor.blue := i * 7168;
      RGBForeColor(theColor);
      SetRect(myRect, myOffScreenWorld^.portRect.left, i * 10,
              myOffScreenWorld^.portRect.right, i * 10 + 10);
      PaintRect(myRect);      {fill offscreen pixel map with gray ramp}
```

```
   END;
   SetGWorld(origPort, origDevice);    {restore onscreen graphics port}
   theColor.red := $0000;
   theColor.green := $0000;
   theColor.blue := $FFFF;
   RGBForeColor(theColor);    {make foreground color all blue}
   theColor.red := $FFFF;
   theColor.green := $0000;
   theColor.blue := $0000;
   RGBBackColor(theColor);    {make background color all red}
      {using blue foreground and red background colors, transfer "gray" }
      { ramp to onscreen graphics port}
   CopyBits(GrafPtr(myOffScreenWorld)^.portBits,   {gray ramp is source}
            GrafPtr(origPort)^.portBits,           {window is destination}
            myOffScreenWorld^.portRect, origPort^.portRect, srcCopy, NIL);
   UnlockPixels(offPixMapHandle);
   DisposeGWorld(myOffScreenWorld);
END;
```

Listing 4-5 uses the `NewGWorld` function, described in the chapter "Offscreen Graphics Worlds," to create an offscreen pixel map. The sample code draws a gray ramp into the offscreen pixel map, which is illustrated on the left side of Plate 2 at the front of this book. Then Listing 4-5 creates an all-blue foreground color and an all-red background color. This sample code then uses the `CopyBits` procedure to transfer the pixels in the offscreen pixel map to the onscreen window, which is shown on the right side of Plate 2.

Here are some hints for using foreground and background colors and the `srcCopy` source mode to color a pixel image:

- You can copy a particular color component of a source pixel without change by setting the foreground color to have a value of $0000 for that component and the background color to have a value of $FFFF for that component. For example, if you want all the pixels in a source image to retain their red values in the destination image, set the red component of the foreground color to $0000, and set the red component of the background color to $FFFF.
- You can invert a particular color component of a source pixel by setting the foreground color to have a value of $FFFF for that component and the background color to have a value of $0000 for that component.
- You can remove a particular color component from all the pixels in the source image by setting the foreground color to have a value of $0000 for that component and the background color to have a value of $0000 for that component.
- You can force a particular color component in all the pixels in the source to be transferred with full intensity by setting the foreground color to have a value of $FFFF for that component and the background color to have a value of $FFFF for that component.

To help make color work well on different screen depths, Color QuickDraw does some validity checking of the foreground and background colors. If your application is drawing to a color graphics port with a pixel depth equal to 1 or 2, and if the foreground and background colors aren't the same but both of them map to the same pixel value, then the foreground color is inverted. This ensures that, for instance, a red image drawn on a green background doesn't map to black on black.

On indexed devices, these source modes produce unexpected colors, because Color QuickDraw performs Boolean operations on the indexes rather than on actual color values, and the resulting index may point to an entirely unrelated color. On direct devices these transfer modes generally do not exhibit rigorous Boolean behavior except when white is set as the background color.

Dithering

With the `CopyBits` and `CopyDeepMask` procedures you can use **dithering,** a technique used by these procedures for mixing existing colors together to create the illusion of a third color that may be unavailable on an indexed device. For example, if you specify dithering when copying a purple image from a 32-bit direct device to an 8-bit indexed device that does not have purple available, these procedures mix blue and red pixels to give the illusion of purple on the 8-bit device.

Dithering is also useful for improving images that you shrink when copying them from a direct device to an indexed device.

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}
```

For example, specifying `srcCopy + ditherCopy` in the `mode` parameter to `CopyBits` causes `CopyBits` to dither the image when it copies the image into the destination pixel map.

Dithering has drawbacks. First, dithering slows the drawing operation. Second, a clipped dithering operation does not provide pixel-for-pixel equivalence to the same unclipped dithering operation. When you don't specify a clipping region, for example, `CopyDeepMask` begins copying the upper-left pixel in your source image and, if necessary, begins calculating how to dither the upper-left pixel and its adjoining pixels in your destination in order to approximate the color of the source pixel. As `CopyDeepMask` continues copying pixels in this manner, there is a cumulative dithering effect based on the preceding pixels in the source image. If you specify a clipping region to `CopyDeepMask`, dithering begins with the upper-left pixel in the clipped region; this ignores the cumulative dithering effect that would otherwise occur by starting at the upper-left corner of the source image. In particular, if you clip and dither a region using the `srcXor` mode, you can't use `CopyDeepMask` a second time to copy that region back into the destination pixel map in order to erase that region.

If you replace the Color Manager's color search function with your own search function (as described in the chapter "Color Manager" in *Inside Macintosh: Advanced Color Imaging*), `CopyBits` and `CopyDeepMask` cannot perform dithering. Without dithering, your application does color mapping on a pixel-by-pixel basis. If your source pixel map is composed of indexed pixels, and if you have installed a custom color search function, Color QuickDraw calls your function once for each color in the color table for the source `PixMap` record. If your source pixel map is composed of direct pixels, Color QuickDraw calls your custom search function for each color in the pixel image for the source `PixMap` record; with an image of many colors, this can take a long time.

If you specify a destination rectangle that is smaller than the source rectangle when using `CopyBits`, `CopyMask`, or `CopyDeepMask` on a direct device, Color QuickDraw automatically uses an averaging technique to produce the destination pixels, maintaining high-quality images when shrinking them. On indexed devices, Color QuickDraw averages these pixels only if you specify dithering. Using dithering even when shrinking 1-bit images can produce much better representations of the original images. (The chapter "QuickDraw Drawing" includes a code sample called `MyShrinkImages`, shown in Listing 3-11 on page 3-33, that illustrates how to use `CopyBits` to scale a bit image when copying it from one window into another.)

Arithmetic Transfer Modes

In addition to the Boolean source modes described previously, Color QuickDraw offers a set of transfer modes that perform arithmetic operations on the values of the red, green, and blue components of the source and destination pixels. Although rarely used by applications, these **arithmetic transfer modes** produce predictable results on indexed devices because they work with RGB colors rather than with color table indexes. These arithmetic transfer modes are represented by the following constants:

```
CONST
   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 a 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}
```


Note

You can use the arithmetic modes for all drawing operations; that is, your application can pass them in parameters to the `PenMode`, `CopyBits`, `CopyDeepMask`, and `TextMode` routines. (The `TextMode` procedure is described in *Inside Macintosh: Text*. ◆

When you use the arithmetic transfer modes, each drawing routine converts indexed source and destination pixels to their RGB components; performs the arithmetic operation on each pair of red, green, and blue components to provide a new RGB color for the destination pixel; and then assigns the destination a pixel value close to the calculated RGB color.

For indexed pixels, the arithmetic transfer modes obtain the full 48-bit RGB color from the CLUT. For direct pixels, the arithmetic transfer modes use the 15 or 24 bits of the truncated RGB color. Note, however, that because the colors for indexed pixels depend on the set of colors currently loaded into a graphics device's CLUT, arithmetic transfer modes may produce effects that differ between indexed and direct devices.

Note
The arithmetic transfer modes have no coloration effects. ◆

When you use the `addPin` mode in a basic graphics port, the maximum allowable value for the destination pixel is always white. In a color graphics port, you can assign the maximum allowable value with the `OpColor` procedure, described on page 4-78. Note that the `addOver` mode is slightly faster than the `addPin` mode.

When you use the `subPin` mode in a basic graphics port, the minimum allowable value for the destination pixel is always black. In a color graphics port, you can assign the minimum allowable value with the `OpColor` procedure. Note that the `subOver` mode is slightly faster than the `subPin` mode.

When you use the `addMax` and `adMin` modes, Color QuickDraw compares each RGB component of the source and destination pixels independently, so the resulting color isn't necessarily either the source or the destination color.

When you use the `blend` mode, Color QuickDraw uses this formula to calculate the weighted average of the source and destination pixels, which Color QuickDraw assigns to the destination pixel:

dest = source × weight / 65,535 + destination × (1 – weight / 65,535)

In this formula, *weight* is an unsigned value between 0 and 65,535, inclusive. In a basic graphics port, the weight is set to 50 percent gray, so that equal weights of the source and destination RGB components are combined to produce the destination color. In a color graphics port, the weight is an `RGBColor` record that individually specifies the weights of the red, green, and blue components. You can assign the weight value with the `OpColor` procedure.

The `transparent` mode is most useful on indexed devices, which have 8-bit and 4-bit pixel depths, and on black-and-white devices. You can specify the `transparent` mode in the `mode` parameter to the `TextMode`, `PenMode`, and `CopyBits` routines. To specify a transparent pattern, add the `transparent` constant to the `patCopy` constant:

```
transparent + patCopy
```

The `transparent` mode is optimized to handle source bitmaps with large transparent holes, as an alternative to specifying an unusual clipping region or mask to the `CopyMask` procedure. Patterns aren't optimized, and may not draw as quickly.

The arithmetic transfer modes are most useful in direct and 8-bit indexed pixels, but work on 4-bit and 2-bit pixels as well. If the destination pixel map is 1 bit deep, the arithmetic transfer mode reverts to a comparable Boolean transfer mode, as shown in Table 4-2. (The `hilite` mode is explained in the next section.)

Table 4-2 Arithmetic modes in a 1-bit environment

Initial arithmetic mode	Resulting source mode
`blend`	`srcCopy`
`addOver`, `subOver`, `hilite`	`srcXor`
`addPin`, `addMax`	`srcBic`
`subPin`, `adMin`, `transparent`	`srcOr`

Because drawing with the arithmetic modes uses the closest matching colors, and not necessarily exact matches, these modes might not produce the results you expect. For instance, suppose your application uses the `srcCopy` mode to paint a green pixel on a screen with 4-bit pixel values. Of the 16 colors available, the closest green may contain a small amount of red, as in RGB components of 300 red, 65,535 green, and 0 blue. Then, your application uses `addOver` mode to paint a red pixel on top of the green pixel, ideally resulting in a yellow pixel. But the red pixel's RGB components are 65,535 red, 0 green, and 0 blue. Adding the red components of the red and green pixels wraps to 300, since the largest representable value is 65,535. In this case, `addOver` causes no visible change at all. You can prevent the maximum value from wrapping around by using the `OpColor` procedure to set the maximum allowable color to white, in which the maximum red value is 65,535. Then you can use the `addPin` mode to produce the desired yellow result.

Note that the arithmetic transfer modes don't call the Color Manager when mapping a requested RGB color to an indexed pixel value. If your application replaces the Color Manager's color-matching routines (which are described in the chapter "Color Manager" in *Inside Macintosh: Advanced Color Imaging*), you must not use these modes, or you must maintain the inverse table yourself.

Highlighting

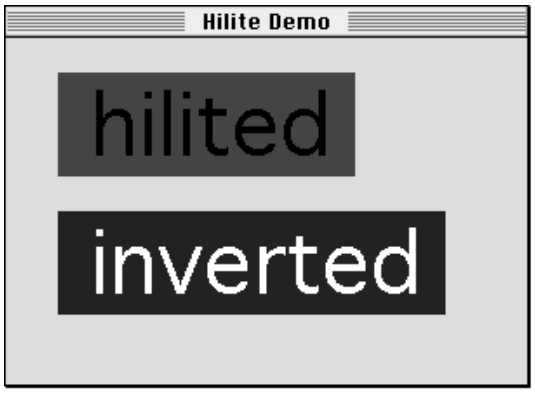
When **highlighting,** Color QuickDraw replaces the background color with the highlight color when your application draws or copies images between graphics ports. This has the visual effect of using a highlighting pen to select the object. For instance, TextEdit (described in *Inside Macintosh: Text*) uses highlighting to indicated selected text; if the highlight color is yellow, TextEdit draws the selected text, then uses `InvertRgn` to produce a yellow background for the text.

With basic QuickDraw, you can use `InvertRect`, `InvertRgn`, `InvertArc`, `InvertRoundRect`, or `InvertPoly` and any image-copying routine that uses the `srcXor` source mode to invert objects on the screen.

In general, however, you should use highlighting with Color QuickDraw when selecting and deselecting objects such as text or graphics. (Highlighting has no effect in basic QuickDraw.) The line reading "hilited" in Figure 4-15 uses highlighting; the user selected red as the highlight color, which the application uses as the background for the text. (This figure shows the effect in grayscale.) The application simply inverts the background for the line reading "inverted." Inversion 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.

Figure 4-15 Difference between highlighting and inverting



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 you can override this by using the `HiliteColor` procedure, described on page 4-78.

To turn highlighting on when using Color QuickDraw, you can clear the highlight bit just before calling `InvertRect`, `InvertRgn`, `InvertArc`, `InvertRoundRect`, `InvertPoly`, or any drawing or image-copying routine that uses the `patXor` or `srcXor` transfer mode. On a bitmap or a 1-bit pixel map, this works exactly like inversion and is compatible with all versions of QuickDraw.

The following constant represents the highlight bit:

```
CONST pHiliteBit = 0; {flag bit in HiliteMode used with BitClr}
```

You can use the `BitClr` procedure as shown in Listing 4-6 to clear system software's highlight bit (`BitClr` is described in *Inside Macintosh: Operating System Utilities*).

Listing 4-6 Setting the highlight bit

```
PROCEDURE MySetHiliteMode;
BEGIN
   BitClr(Ptr(HiliteMode), pHiliteBit);
END;
```

Listing 4-7 shows the code that produced the effects in Figure 4-15.

Listing 4-7 Using highlighting for text

```
PROCEDURE HiliteDemonstration (window: WindowPtr);
CONST
   s1 = ' hilited ';
   s2 = ' inverted ';
VAR
   familyID:   Integer;
   r1, r2:     Rect;
   info:       FontInfo;
   bg:         RGBColor;
BEGIN
   TextSize(48);
   GetFontInfo(info);
   SetRect(r1, 0, 0, StringWidth(s1), info.ascent + info.descent);
   SetRect(r2, 0, 0, StringWidth(s2), info.ascent + info.descent);
   OffsetRect(r1, 30, 20);
   OffsetRect(r2, 30, 100);
{fill the background with a light-blue color}
   bg.red := $A000;
   bg.green := $FFFF;
   bg.blue := $E000;
   RGBBackColor(bg);
   EraseRect(window^.portRect);
{draw the string to highlight}
   MoveTo(r1.left + 2, r1.bottom - info.descent);
   DrawString(s1);
   MySetHiliteMode;  {clear the highlight bit}
{InvertRect replaces pixels in background color with the }
{ user-specified highlight color}
   InvertRect(r1);
{the highlight bit is reset automatically}
{show inverted text, for comparison}
   MoveTo(r2.left + 2, r2.bottom - info.descent);
   DrawString(s2);
   InvertRect(r2);
END;
```

Color QuickDraw resets the highlight bit after performing each drawing operation, so your application should always clear the highlight bit immediately before calling a routine with which you want to use highlighting.

Another way to use highlighting is to add this constant or its value to the mode you specify to the `PenMode`, `CopyBits`, `CopyDeepMask`, and `TextMode` routines:

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

Highlighting uses the pattern or source image to decide which bits to exchange; only bits that are on in the pattern or source image can be highlighted in the destination.

A very small selection should probably not use highlighting, because it might be too hard to see the selection in the highlight color. TextEdit, for instance, uses highlighting to select and deselect text, but not to highlight the insertion point.

Highlighting is optimized to look for consecutive pixels in either the highlight or background colors. For example, if the source is an all-black pattern, the highlighting is especially fast, operating internally on one long word at a time instead of one pixel at a time. Highlighting a large area without such consecutive pixels (a gray pattern, for instance) can be slow.

Color QuickDraw Reference

This section describes the data structures, routines, and resources that are specific to Color QuickDraw.

"Data Structures" shows the Pascal data structures for the `PixMap`, `CGrafPort`, `RGBColor`, `ColorSpec`, `ColorTable`, `MatchRec`, `PixPat`, `CQDProcs`, and `GrafVars` records.

"Color QuickDraw Routines" describes routines for creating and closing color graphics ports, managing a color graphics pen, changing the background pixel pattern, drawing with Color QuickDraw colors, determining current colors and best intermediate colors, calculating color fills, creating and disposing of pixel maps, creating and disposing of pixel patterns, creating and disposing of color tables, customizing Color QuickDraw operations, and reporting changes to QuickDraw data structures that applications typically shouldn't make. "Application-Defined Routine" describes how to write your own color search function for customizing the `SeedCFill` and `CalcCMask` procedures.

"Resources" describes the pixel pattern resource, the color table resource, and the color icon resource.

Data Structures

This section shows the Pascal data structures for the `PixMap`, `CGrafPort`, `RGBColor`, `ColorSpec`, `ColorTable`, `MatchRec`, `PixPat`, `CQDProcs`, and `GrafVars` records.

Analogous to the bitmap that basic QuickDraw uses to describe a bit image, a pixel map is used by Color QuickDraw to describe a pixel image. A pixel map, which is a data structure of type `PixMap`, contains information about the dimensions and contents of a pixel image, as well as information about the image's storage format, depth, resolution, and color usage.

As a basic graphics port (described in the chapter "Basic QuickDraw") defines the black-and-white and basic eight-color drawing environment for basic QuickDraw, a color graphics port defines the more sophisticated color drawing environment for Color QuickDraw. A color graphics port is defined by a data structure of type `CGrafPort`.

You usually specify a color to Color QuickDraw by creating an `RGBColor` record in which you assign the red, green, and blue values of the color. For example, when you want to set the foreground color for drawing, you create an `RGBColor` record that defines the foreground color you desire, then you pass that record as a parameter to the `RGBForeColor` procedure.

When creating a `PixMap` record for an indexed device, Color QuickDraw creates a `ColorTable` record that defines the best colors available for the pixel image on that graphics device. The Color Manager also stores a `ColorTable` record for the currently available colors in the graphics device's CLUT.

One of the fields in a `ColorTable` record requires a value of type `cSpecArray`, which is defined as an array of `ColorSpec` records. Typically, your application needs to create `ColorTable` records and `ColorSpec` records only if it uses the Palette Manager, as described in the chapter "Palette Manager" in *Inside Macintosh: Advanced Color Imaging*.

You can customize the `SeedCFill` and `CalcCMask` procedures by writing your own color search functions and pointing to them in the `matchProc` parameters for these procedures. When `SeedCFill` or `CalcCMask` calls your color search function, the `GDRefCon` field of the current `GDevice` record (described in the chapter "Graphics Devices") contains a pointer to a `MatchRec` record. This record contains the RGB value of the seed pixel or seed color for which your color search function should search.

Your application typically does not create `PixPat` records. Although you can create `PixPat` records in your program code, it is usually easier to create pixel patterns using the pixel pattern resource, which is described on page 4-103.

You need to use the `CQDProcs` record only if you customize one or more of QuickDraw's low-level drawing routines.

Finally, the `GrafVars` record contains color information that supplements the information in the `CGrafPort` record, of which it is logically a part.

PixMap

A pixel map, which is defined by a data structure of type `PixMap`, contains information about the dimensions and contents of a pixel image, as well as information on the image's storage format, depth, resolution, and color usage.

```
TYPE  PixMap =
RECORD
   baseAddr:      Ptr;        {pixel image}
   rowBytes:      Integer;    {flags, and row width}
   bounds:        Rect;       {boundary rectangle}
   pmVersion:     Integer;    {PixMap record version number}
   packType:      Integer;    {packing format}
   packSize:      LongInt;    {size of data in packed state}
   hRes:          Fixed;      {horizontal resolution}
   vRes:          Fixed;      {vertical resolution}
   pixelType:     Integer;    {format of pixel image}
   pixelSize:     Integer;    {physical bits per pixel}
   cmpCount:      Integer;    {logical components per pixel}
   cmpSize:       Integer;    {logical bits per component}
   planeBytes:    LongInt;    {offset to next plane}
   pmTable:       CTabHandle; {handle to the ColorTable record }
                              { for this image}
   pmReserved:    LongInt;    {reserved for future expansion}
END;
```

Field descriptions

`baseAddr` For an onscreen pixel image, a pointer to the first byte of the image. For optimal performance, this should be a multiple of 4. The pixel image that appears on a screen is normally stored on a graphics card rather than in main memory.

▲ **WARNING**
The `baseAddr` field of the `PixMap` record for an offscreen graphics world contains a handle instead of a pointer. You must use the `GetPixBaseAddr` function (described in the chapter "Offscreen Graphics Worlds" in this book) to obtain a pointer to the `PixMap` record for an offscreen graphics world. Your application should never directly access the `baseAddr` field of the `PixMap` record for an offscreen graphics world; instead, your application should always use `GetPixBaseAddr`. ▲

`rowBytes`	The offset in bytes from one row of the image to the next. The value must be even, less than $4000, and for best performance it should be a multiple of 4. The high 2 bits of `rowBytes` are used as flags. If bit 15 = 1, the data structure pointed to is a `PixMap` record; otherwise it is a `BitMap` record.
`bounds`	The boundary rectangle, which links the local coordinate system of a graphics port to QuickDraw's global coordinate system and defines the area of the bit image into which QuickDraw can draw. By default, the boundary rectangle is the entire main screen. Do not use the value of this field to determine the size of the screen; instead use the value of the `gdRect` field of the `GDevice` record for the screen, as described in the chapter "Graphics Devices" in this book.
`pmVersion`	The version number of Color QuickDraw that created this `PixMap` record. The value of `pmVersion` is normally 0. If `pmVersion` is 4, Color QuickDraw treats the `PixMap` record's `baseAddr` field as 32-bit clean. (All other flags are private.) Most applications never need to set this field.
`packType`	The packing algorithm used to compress image data. Color QuickDraw currently supports a `packType` of 0, which means no packing, and values of 1 to 4 for packing direct pixels.
`packSize`	The size of the packed image in bytes. When the `packType` field contains the value 0, this field is always set to 0.
`hRes`	The horizontal resolution of the pixel image in pixels per inch. This value is of type `Fixed`; by default, the value here is $00480000 (for 72 pixels per inch).
`vRes`	The vertical resolution of the pixel image in pixels per inch. This value is of type `Fixed`; by default, the value here is $00480000 (for 72 pixels per inch).
`pixelType`	The storage format for a pixel image. Indexed pixels are indicated by a value of 0. Direct pixels are specified by a value of `RGBDirect`, or 16. In the `PixMap` record of the `GDevice` record (described in the chapter "Graphics Devices") for a direct device, this field is set to the constant `RGBDirect` when the screen depth is set.
`pixelSize`	Pixel depth; that is, the number of bits used to represent a pixel. Indexed pixels can have sizes of 1, 2, 4, and 8 bits; direct pixel sizes are 16 and 32 bits.
`cmpCount`	The number of components used to represent a color for a pixel. With indexed pixels, each pixel is a single value representing an index in a color table, and therefore this field contains the value 1—the index is the single component. With direct pixels, each pixel contains three components—one integer each for the intensities of red, green, and blue—so this field contains the value 3.

`cmpSize`	The size in bits of each component for a pixel. Color QuickDraw expects that the sizes of all components are the same, and that the value of the `cmpCount` field multiplied by the value of the `cmpSize` field is less than or equal to the value in the `pixelSize` field. For an indexed pixel value, which has only one component, the value of the `cmpSize` field is the same as the value of the `pixelSize` field—that is, 1, 2, 4, or 8. For direct pixels there are two additional possibilities: A 16-bit pixel, which has three components, has a `cmpSize` value of 5. This leaves an unused high-order bit, which Color QuickDraw sets to 0. A 32-bit pixel, which has three components (red, green, and blue), has a `cmpSize` value of 8. This leaves an unused high-order byte, which Color QuickDraw sets to 0. If presented with a 32-bit image—for example, in the `CopyBits` procedure—Color QuickDraw passes whatever bits are there, and it does not set the high byte to 0. Generally, therefore, your application should clear the memory for the image to 0 before creating a 16-bit or 32-bit image. The Memory Manager functions `NewHandleClear` and `NewPtrClear`, described in *Inside Macintosh: Memory*, assist you in allocating prezeroed memory.
`planeBytes`	The offset in bytes from one drawing plane to the next. This field is set to 0.
`pmTable`	A handle to a `ColorTable` record (described on page 4-56) for the colors in this pixel map.
`pmReserved`	Reserved for future expansion. This field must be set to 0 for future compatibility.

Note that the pixel map for a window's color graphics port always consists of the pixel depth, color table, and boundary rectangle of the main screen, even if the window is created on or moved to an entirely different screen.

CGrafPort

A color graphics port, which is defined by a data structure of type `CGrafPort`, defines a complete drawing environment that determines where and how color graphics operations take place.

All graphics operations are performed in graphics ports. Before a color graphics port can be used, it must be allocated and initialized with the `OpenCPort` procedure, which is described on page 4-64. Normally, you don't call `OpenCPort` yourself. In most cases your application draws into a color window you've created with the `GetNewCWindow` or `NewCWindow` function or draws into an offscreen graphics world created with the `NewGWorld` function. The two Window Manager functions (described in the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*) and the

NewGWorld function (described in the chapter "Offscreen Graphics Worlds" in this book) call OpenCPort to create the window's graphics port.

You can have many graphics ports open at once; each one has its own local coordinate system, pen pattern, background pattern, pen size and location, font and font style, and pixel map in which drawing takes place.

Several fields in this record define your application's drawing area. All drawing in a graphics port occurs in the intersection of the graphics port's boundary rectangle and its port rectangle. Within that intersection, all drawing is cropped to the graphics port's visible region and its clipping region.

The Window Manager and Dialog Manager routines GetNewWindow, GetNewDialog, Alert, StopAlert, NoteAlert, and CautionAlert (described in *Inside Macintosh: Macintosh Toolbox Essentials*) create a color graphics port if color-aware resources (such as resource types 'wctb', 'dctb', or 'actb') are present.

The CGrafPort record is the same size as the GrafPort record, and most of its fields are identical. The structure of the CGrafPort record, is as follows:

```
TYPE CGrafPtr  = ^CGrafPort;
CGrafPort =
RECORD
   device:        Integer;       {device ID for font selection}
   portPixMap:    PixMapHandle;  {handle to PixMap record}
   portVersion:   Integer;       {highest 2 bits always set}
   grafVars:      Handle;        {handle to a GrafVars record}
   chExtra:       Integer;       {added width for nonspace characters}
   pnLocHFrac:    Integer;       {pen fraction}
   portRect:      Rect;          {port rectangle}
   visRgn:        RgnHandle;     {visible region}
   clipRgn:       RgnHandle;     {clipping region}
   bkPixPat:      PixPatHandle;  {background pattern}
   rgbFgColor:    RGBColor;      {requested foreground color}
   rgbBkColor:    RGBColor;      {requested background color}
   pnLoc:         Point;         {pen location}
   pnSize:        Point;         {pen size}
   pnMode:        Integer;       {pattern mode}
   pnPixPat:      PixPatHandle;  {pen pattern}
   fillPixPat:    PixPatHandle;  {fill pattern}
   pnVis:         Integer;       {pen visibility}
   txFont:        Integer;       {font number for text}
   txFace:        Style;         {text's font style}
   txMode:        Integer;       {source mode for text}
   txSize:        Integer;       {font size for text}
   spExtra:       Fixed;         {added width for space characters}
   fgColor:       LongInt;       {actual foreground color}
```

```
   bkColor:       LongInt;       {actual background color}
   colrBit:       Integer;       {plane being drawn}
   patStretch:    Integer;       {used internally}
   picSave:       Handle;        {picture being saved, used internally}
   rgnSave:       Handle;        {region being saved, used internally}
   polySave:      Handle;        {polygon being saved, used internally}
   grafProcs:     CQDProcsPtr;   {low-level drawing routines}
END;
```

▲ **WARNING**

You can read the fields of a `CGrafPort` record directly, but you should not store values directly into them. Use the QuickDraw routines described in this book to alter the fields of a graphics port. ▲

Field descriptions

`device` Device-specific information that's used by the Font Manager to achieve the best possible results when drawing text in the graphics port. There may be physical differences in the same logical font for different output devices, to ensure the highest-quality printing on the device being used. For best results on the screen, the default value of the device field is 0.

`portPixMap` A handle to a `PixMap` record (described on page 4-46), which describes the pixels in this color graphics port.

`portVersion` In the highest 2 bits, flags set to indicate that this is a `CGrafPort` record and, in the remainder of the field, the version number of Color QuickDraw that created this record.

`grafVars` A handle to the `GrafVars` record (described on page 4-62), which contains additional graphics fields of color information.

`chExtra` A fixed-point number by which to widen every character, excluding the space character, in a line of text. This value is used in proportional spacing. The value in this field is in 4.12 fractional notation: 4 bits of signed integer are followed by 12 bits of fraction. This value is multiplied by the value in the `txSize` field before it is used. By default, this field contains the value 0.

`pnLocHFrac` The fractional horizontal pen position used when drawing text. The value in this field represents the low word of a `Fixed` number; in decimal, its initial value is 0.5.

`portRect`	The port rectangle that defines a subset of the pixel map to be used for drawing. All drawing done by the application occurs inside the port rectangle. (In a window's graphics port, the port rectangle is also called the *content region*.) The port rectangle uses the local coordinate system defined by the boundary rectangle in the `portPixMap` field of the `PixMap` record. The upper-left corner (which for a window is called the *window origin*) of the port rectangle usually has a vertical coordinate of 0 and a horizontal coordinate of 0, although you can use the `SetOrigin` procedure (described in the chapter "Basic QuickDraw") to change the coordinates of the window origin. The port rectangle usually falls within the boundary rectangle, but it's not required to do so.
`visRgn`	The region of the graphics port that's actually visible on the screen—that is, the part of the window that's not covered by other windows. By default, the visible region is equivalent to the port rectangle. The visible region has no effect on images that aren't displayed on the screen.
`clipRgn`	The graphics port's clipping region, an arbitrary region that you can use to limit drawing to any region within the port rectangle. The default clipping region is set arbitrarily large; using the `ClipRect` procedure (described in the chapter "Basic QuickDraw"), you have full control over its setting. Unlike the visible region, the clipping region affects the image even if it isn't displayed on the screen.
`bkPixPat`	A handle to a `PixPat` record (described on page 4-58) that describes the background pixel pattern. Procedures such as `ScrollRect` (described in the chapter "Basic QuickDraw") and `EraseRect` (described in the chapter "QuickDraw Drawing") use this pattern for filling scrolled or erased areas. Your application can use the `BackPixPat` procedure (described on page 4-69) to change the background pixel pattern.
`rgbFgColor`	An `RGBColor` record (described on page 4-55) that contains the requested foreground color. By default, the foreground color is black, but you can use the `RGBForeColor` procedure (described on page 4-70) to change the foreground color.
`rgbBkColor`	An `RGBColor` record that contains the requested background color. By default, the background color is white, but you can use the `RGBBackColor` procedure (described on page 4-72) to change the background color.

`pnLoc`	The point where QuickDraw will begin drawing the next line, shape, or character. It can be anywhere on the coordinate plane; there are no restrictions on the movement or placement of the pen. The location of the graphics pen is a point in the graphics port's coordinate system, not a pixel in a pixel 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. The graphics pen is described in detail in the chapter "QuickDraw Drawing."
`pnSize`	The vertical height and horizontal width of the graphics pen. The default size is a 1-by-1 pixel square; the vertical height and horizontal width 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. You can use the `PenSize` procedure (described in the chapter "QuickDraw Drawing") to change the value in this field.
`pnMode`	The pattern mode—that is, a Boolean operation that determines the how Color QuickDraw transfers the pen pattern to the pixel map during drawing operations. When the graphics pen draws into a pixel map, Color QuickDraw first determines what pixels in the pixel image are affected and finds their corresponding pixels in the pen pattern. Color QuickDraw then does a pixel-by-pixel comparison based on the pattern mode, which specifies one of eight Boolean transfer operations to perform. Color QuickDraw stores the resulting pixel in its proper place in the image. Pattern modes for a color graphics port are described in "Boolean Transfer Modes With Color Pixels" beginning on page 4-32.
`pnPixPat`	A handle to a `PixPat` record (described on page 4-58) that describes a pixel pattern used like the ink in the graphics pen. Color QuickDraw uses the pixel pattern defined in the `PixPat` record when you use the `Line` and `LineTo` procedures to draw lines with the pen, framing procedures such as `FrameRect` to draw shape outlines with the pen, and painting procedures such as `PaintRect` to paint shapes with the pen.
`fillPixPat`	A handle to a `PixPat` record (described on page 4-58) that describes the pixel pattern that's used when you call a procedure such as `FillRect` to fill an area. Notice that this is not in the same location as the `fillPat` field in a `GrafPort` record.
`pnVis`	The graphics pen's visibility—that is, whether it draws on the screen. The graphics pen is described in detail in the chapter "QuickDraw Drawing."

`txFont`	A font number that identifies the font to be used in the graphics port. The font number 0 represents the system font. (A font is defined as a collection of images that represent the individual characters of the font.) Fonts are described in detail in *Inside Macintosh: Text*.
`txFace`	The character style of the text, with values from the set defined by the `Style` data type, which includes such styles as bold, italic, and shaded. You can apply stylistic variations either alone or in combination. Character styles are described in detail in *Inside Macintosh: Text*.
`txMode`	One of three Boolean source modes that determines the way characters are placed in the bit image. This mode functions much like a pattern mode specified in the `pnMode` field: when drawing a character, Color QuickDraw determines which pixels in the image are affected, does a pixel-by-pixel comparison based on the mode, and stores the resulting pixels in the image. Only three source modes—`srcOr`, `srcXor`, and `srcBic`—should be used for drawing text. See the chapter "QuickDraw Text" in *Inside Macintosh: Text* for more information about QuickDraw's text-handling capabilities.
`txSize`	The text size in pixels. The Font Manager uses this information to provide the bitmaps for text drawing. (The Font Manager is described in detail in the chapter "Font Manager" in *Inside Macintosh: Text*.) The value in this field can be represented by point size × device resolution / 72 dpi where *point* is a typographical term meaning approximately 1/72 inch.
`spExtra`	A fixed-point number equal to the average number of pixels by which each space character should be widened to fill out the line. The `spExtra` field is useful when a line of characters is to be aligned with both the left and the right margin (sometimes called *full justification*).
`fgColor`	The pixel value of the foreground color supplied by the Color Manager. This is the best available approximation in the CLUT to the color specified in the `rgbFgColor` field.
`bkColor`	The pixel value of the background color supplied by the Color Manager. This is the best available approximation in the CLUT to the color specified in the `rgbBkColor` field.
`colrBit`	Reserved.
`patStretch`	A value used during output to a printer to expand patterns if necessary. Your application should not change this value.

`picSave`	The state of the picture definition. If no picture is open, this field contains `NIL`; otherwise it contains a handle to information related to the picture definition. Your application shouldn't be concerned about exactly what information the handle leads to; you may, however, save the current value of this field, set the field to `NIL` to disable the picture definition, and later restore it to the saved value to resume defining the picture. Pictures are described in the chapter "Pictures" in this book.
`rgnSave`	The state of the region definition. If no region is open, this field contains `NIL`; otherwise it contains a handle to information related to the region definition. Your application shouldn't be concerned about exactly what information the handle leads to; you may, however, save the current value of this field, set the field to `NIL` to disable the region definition, and later restore it to the saved value to resume defining the region.
`polySave`	The state of the polygon definition. If no polygon is open, this field contains `NIL`; otherwise it contains a handle to information related to the polygon definition. Your application shouldn't be concerned about exactly what information the handle leads to; you may, however, save the current value of this field, set the field to `NIL` to disable the polygon definition, and later restore it to the saved value to resume defining the polygon.
`grafProcs`	An optional pointer to a `CQDProcs` record (described on page 4-60) that your application can store into if you want to customize Color QuickDraw drawing routines or use Color QuickDraw in other advanced, highly specialized ways.

All Color QuickDraw operations refer to a graphics port by a pointer defined by the data type `CGrafPtr`. (For historical reasons, a graphics port is one of the few objects in the Macintosh system software that's referred to by a pointer rather than a handle.) All Window Manager routines that accept a window pointer also accept a pointer to a color graphics port.

Your application should never need to directly change the fields of a `CGrafPort` record. If you find it absolutely necessary for your application to so, immediately use the `PortChanged` procedure to notify Color QuickDraw that your application has changed the `CGrafPort` record. The `PortChanged` procedure is described on page 4-99.

RGBColor

You usually specify a color to Color QuickDraw by creating an `RGBColor` record in which you assign the red, green, and blue values of the color. For example, when you want to set the foreground color for drawing, you create an `RGBColor` record that defines the foreground color you desire; then you pass that record as a parameter to the `RGBForeColor` procedure.

In an `RGBColor` record, three 16-bit unsigned integers give the intensity values for the three additive primary colors.

```
TYPE RGBColor =
RECORD
   red:     Integer;    {red component}
   green:   Integer;    {green component}
   blue:    Integer;    {blue component}
END;
```

Field descriptions

`red`	An unsigned integer specifying the red value of the color.
`green`	An unsigned integer specifying the green value of the color.
`blue`	An unsigned integer specifying the blue value of the color.

ColorSpec

When creating a `PixMap` record (described on page 4-46) for an indexed device, Color QuickDraw creates a `ColorTable` record that defines the best colors available for the pixel image on that graphics device. The Color Manager also stores a `ColorTable` record for the currently available colors in the graphics device's CLUT.

One of the fields in a `ColorTable` record requires a value of type `cSpecArray`, which is defined as an array of `ColorSpec` records. Typically, your application never needs to create `ColorTable` records or `ColorSpec` records. For completeness, the data structure of type `ColorSpec` is shown here, and the data structure of type `ColorTable` is shown next.

```
TYPE
cSpecArray: ARRAY[0..0] Of ColorSpec;
ColorSpec =
RECORD
   value:   Integer;    {index or other value}
   rgb:     RGBColor;   {true color}
END;
```

Field descriptions

`value` The pixel value assigned by Color QuickDraw for the color specified in the `rgb` field of this record. Color QuickDraw assigns a pixel value based on the capabilities of the user's screen. For indexed devices, the pixel value is an index number assigned by the Color Manager to the closest color available on the indexed device; for direct devices, this value expresses the best available red, green, and blue values for the color on the direct device.

`rgb` An `RGBColor` record (described in the previous section) that fully specifies the color whose approximation Color QuickDraw specifies in the `value` field.

ColorTable

When creating a `PixMap` record (described on page 4-46) for a particular graphics device, Color QuickDraw creates a `ColorTable` record that defines the best colors available for the pixel image on that particular graphics device. The Color Manager also creates a `ColorTable` record of all available colors for use by the CLUT on indexed devices.

Typically, your application needs to create `ColorTable` records only if it uses the Palette Manager, as described in the chapter "Palette Manager" in *Inside Macintosh: Advanced Color Imaging*. The data structure of type `ColorTable` is shown here.

```
TYPE CTabHandle   = ^CTabPtr;
CTabPtr           = ^ColorTable;
ColorTable        =
RECORD
   ctSeed:  LongInt;    {unique identifier from table}
   ctFlags: Integer;    {flags describing the value in the }
                        { ctTable field; clear for a pixel map}
   ctSize:  Integer;    {number of entries in the next field }
                        { minus 1}
   ctTable: cSpecArray; {an array of ColorSpec records}
END;
```

Field descriptions

`ctSeed` Identifies a particular instance of a color table. The Color Manager uses the `ctSeed` value to compare an indexed device's color table with its associated inverse table (a table it uses for fast color lookup). When the color table for a graphics device has been changed, the Color Manager needs to rebuild the inverse table. See the chapter "Color Manager" in *Inside Macintosh: Advanced Color Imaging* for more information on inverse tables.

`ctFlags`	Flags that distinguish pixel map color tables from color tables in `GDevice` records (which are described in the chapter "Graphics Devices" in this book).
`ctSize`	One less than the number of entries in the table.
`ctTable`	An array of `ColorSpec` entries, each containing a pixel value and a color specified by an `RGBColor` record, as described in the previous section.

Your application should never need to directly change the fields of a `ColorTable` record. If you find it absolutely necessary for your application to so, immediately use the `CTabChanged` procedure to notify Color QuickDraw that your application has changed the `ColorTable` record. The `CTabChanged` procedure is described on page 4-97.

MatchRec

As described in "Application-Defined Routine" on page 4-101, you can customize the `SeedCFill` and `CalcCMask` procedures by writing your own color search functions and pointing to them in the `matchProc` parameters for these procedures.

When `SeedCFill` or `CalcCMask` calls your color search function, the `GDRefCon` field of the current `GDevice` record (described in the chapter "Graphics Devices") contains a pointer to a `MatchRec` record. This record contains the RGB value of the seed pixel or seed color for which your color search function should search. This record has the following structure:

```
MatchRec =
RECORD
   red:        Integer; {red component of seed}
   green:      Integer; {green component of seed}
   blue:       Integer; {blue component of seed}
   matchData:  LongInt; {value in matchData parameter of }
                        { SeedCFill or CalcCMask}
END;
```

Field descriptions

`red`	Red value of the seed.
`green`	Green value of the seed.
`blue`	Blue value of the seed.
`matchData`	The value passed in the `matchData` parameter of the `SeedCFill` or `CalcCMask` procedure.

PixPat

Your application typically does not create `PixPat` records. Although you can create such records in your program code, it is usually easier to create pixel patterns using the pixel pattern resource, which is described on page 4-103.

A `PixPat` record is defined as follows:

```
TYPE PixPatHandle = ^PixPatPtr;
PixPatPtr         = ^PixPat;
PixPat            =
RECORD
   patType:    Integer;       {pattern type}
   patMap:     PixMapHandle;  {pattern characteristics}
   patData:    Handle;        {pixel image defining pattern}
   patXData:   Handle;        {expanded pixel image}
   patXValid:  Integer;       {flags for expanded pattern data}
   patXMap:    Handle;        {handle to expanded pattern data}
   pat1Data:   Pattern;       {a bit pattern for a GrafPort }
                              { record}
END;
```

Field descriptions

`patType` The pattern's type. The value 0 specifies a basic QuickDraw bit pattern, the value 1 specifies a full-color pixel pattern, and the value 2 specifies an RGB pattern. These pattern types are described in greater detail in the rest of this section.

`patMap` A handle to a `PixMap` record (described on page 4-46) that describes the pattern's pixel image. The `PixMap` record can contain indexed or direct pixels.

`patData` A handle to the pattern's pixel image.

`patXData` A handle to an expanded pixel image used internally by Color QuickDraw.

`patXValid` A flag that, when set to –1, invalidates the expanded data.

`patXMap` Reserved for use by Color QuickDraw.

`pat1Data` A bit pattern (described in the chapter "QuickDraw Drawing") to be used when this pattern is drawn into a `GrafPort` record (described in the chapter "Basic QuickDraw"). The `NewPixPat` function (described on page 4-88) sets this field to 50 percent gray.

When used for a color graphics port, the basic QuickDraw procedures `PenPat` and `BackPat` (described in the chapter "Basic QuickDraw") store pixel patterns in, respectively, the `pnPixPat` and `bkPixPat` fields of the `CGrafPort` record and set the `patType` field of the `PixPat` field to 0 to indicate that the `PixPat` record contains a bit pattern. Such patterns are limited to 8-by-8 pixel dimensions and, instead of being drawn in black and white, are always drawn using the colors specified in the `CGrafPort` record's `rgbFgColor` and `rgbBkColor` fields, respectively.

In a full-color pixel pattern, the `patType` field contains the value 1, and the pattern's dimensions, depth, resolution, set of colors, and other characteristics are defined by a `PixMap` record, referenced by the handle in the `patMap` field of the `PixPat` record. Full-color pixel patterns contain color tables that describe the colors they use. Generally such a color table contains one entry for each color used in the pattern. For instance, if your pattern has five colors, you would probably create a 4 bits per pixel pattern that uses pixel values 0–4, and a color table with five entries, numbered 0–4, that contain the RGB specifications for those pixel values.

However, if you don't specify a color table for a pixel value, Color QuickDraw assigns a color to that pixel value. The largest unassigned pixel value becomes the foreground color; the smallest unassigned pixel value is assigned the background color. Remaining unassigned pixel values are given colors that are evenly distributed between the foreground and background.

For instance, in the color table mentioned above, pixel values 5–15 are unused. Assume that the foreground color is black and the background color is white. Pixel value 15 is assigned the foreground color, black; pixel value 5 is assigned the background color, white; the nine pixel values between them are assigned evenly distributed shades of gray. If the `PixMap` record's color table is set to `NIL`, all pixel values are determined by blending the foreground and background colors.

Full-color pixel patterns are not limited to a fixed size: their height and width can be any power of 2, as specified by the height and width of the boundary rectangle for the `PixMap` record specified in the `patMap` field. A pattern 8 bits wide, which is the size of a bit pattern, has a row width of just 1 byte, contrary to the usual rule that the `rowBytes` field must be even. Read this pattern type into memory using the `GetPixPat` function (described on page 4-88), and set it using the `PenPixPat` or `BackPixPat` procedure (described on page 4-67 and page 4-69, respectively).

The pixel map specified in the `patMap` field of the `PixPat` record defines the pattern's characteristics. The `baseAddr` field of the `PixMap` record for that pixel map is ignored. For a full-color pixel pattern, the actual pixel image defining the pattern is stored in the handle in the `patData` field of the `PixPat` record. The pattern's pixel depth need not match that of the pixel map into which it's transferred; the depth is adjusted automatically when the pattern is drawn. Color QuickDraw maintains a private copy of the pattern's pixel image, expanded to the current screen depth and aligned to the current graphics port, in the `patXData` field of the `PixPat` record.

In an RGB pixel pattern, the `patType` field contains the value 2. Using the `MakeRGBPat` procedure (described on page 4-90), your application can specify the exact color it wants to use. Color QuickDraw selects a pattern to approximate that color. In this way, your application can effectively increase the color resolution of the screen. RGB pixel patterns are particularly useful for dithering: mixing existing colors together to create the illusion of a third color that's unavailable on an indexed device. The `MakeRGBPat` procedure aids in this process by constructing a dithered pattern to approximate a given absolute color. An RGB pixel pattern can display 125 different patterns on a 4-bit screen, or 2197 different patterns on an 8-bit screen.

An RGB pixel pattern has an 8-by-8 pixel pattern that is 2 bits deep. For an RGB pixel pattern, the `RGBColor` record that you specify to the `MakeRGBPat` procedure defines the image; there is no image data.

Your application should never need to directly change the fields of a `PixPat` record. If you find it absolutely necessary for your application to so, immediately use the `PixPatChanged` procedure to notify Color QuickDraw that your application has changed the `PixPat` record. The `PixPatChanged` procedure is described on page 4-98.

CQDProcs

You need to use the `CQDProcs` record only if you customize one or more of QuickDraw's standard low-level drawing routines, which are described in the chapter "QuickDraw Drawing." You can use the `SetStdCProcs` procedure, described on page 4-96, to create a `CQDProcs` record.

```
CQDProcsPtr = ^CQDProcs
CQDProcs    =
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}
   opcodeProc:    Ptr;  {reserved for future use}
   newProc1:      Ptr;  {reserved for future use}
   newProc2:      Ptr;  {reserved for future use}
```

```
   newProc3:      Ptr;  {reserved for future use}
   newProc4:      Ptr;  {reserved for future use}
   newProc5:      Ptr;  {reserved for future use}
   newProc6:      Ptr;  {reserved for future use}
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 `StdTxMeas` 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.
`opcodeProc`	Reserved for future use.
`newProc1`	Reserved for future use.
`newProc2`	Reserved for future use.
`newProc3`	Reserved for future use.
`newProc4`	Reserved for future use.
`newProc5`	Reserved for future use.
`newProc6`	Reserved for future use.

GrafVars

The `GrafVars` record contains color information in addition to that in the `CGrafPort` record, of which it is logically a part; the information is used by Color QuickDraw and the Palette Manager.

```
TYPE GrafVars =
RECORD
   rgbOpColor:       RGBColor;   {color for addPin, subPin, and }
                                 { blend}
   rgbHiliteColor:   RGBColor;   {color for highlighting}
   pmFgColor:        Handle;     {palette handle for foreground }
                                 { color}
   pmFgIndex:        Integer;    {index value for foreground}
   pmBkColor:        Handle;     {palette handle for background }
                                 { color}
   pmBkIndex:        Integer;    {index value for background}
   pmFlags:          Integer;    {flags for Palette Manager}
END;
```

Field descriptions

`rgbOpColor` The color for the arithmetic transfer operations `addPin`, `subPin`, and `blend`.

`rgbHiliteColor`
The highlight color for this graphics port.

`pmFgColor` A handle to the palette that contains the foreground color.

`pmFgIndex` The index value into the palette for the foreground color.

`pmBkColor` A handle to the palette that contains the background color.

`pmBkIndex` The index value into the palette for the background color.

`pmFlags` Flags private to the Palette Manager.

See the chapter “Palette Manager” in *Inside Macintosh: Advanced Color Imaging* for further information on how the Palette Manager handles colors in a color graphics port.

Color QuickDraw Routines

This section describes Color QuickDraw's routines for creating and closing color graphics ports, managing a color graphics pen, changing the background pixel pattern, drawing with Color QuickDraw colors, determining current colors and best intermediate colors, calculating color fills, creating and disposing of pixel maps, creating and disposing of pixel patterns, creating and disposing of color tables, customizing Color QuickDraw operations, and reporting to QuickDraw that your application has directly changed those data structures that applications generally shouldn't manipulate.

To initialize Color QuickDraw, use the `InitGraf` procedure, described in the chapter "Basic QuickDraw." Besides initializing basic QuickDraw, this procedure initializes Color QuickDraw on computers that support it.

In addition to `InitGraf`, all other basic QuickDraw routines work with Color QuickDraw. For example, you can use the `GetPort` procedure to save the current color graphics port, and you can use the `CopyBits` procedure to copy an image between two different color graphics ports. See the chapters "Basic QuickDraw" and "QuickDraw Drawing" for descriptions of additional routines that you can use with Color QuickDraw.

Opening and Closing Color Graphics Ports

All graphics operations are performed in graphics ports. Before a color graphics port can be used, it must be allocated with the `OpenCPort` procedure and initialized with the `InitCPort` procedure. Normally, your application does not call these procedures directly. Instead, your application creates a graphics port by using the `GetNewCWindow` or `NewCWindow` function (described in the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*) or the `NewGWorld` function (described in the chapter "Offscreen Graphics Worlds" in this book). These functions automatically call `OpenCPort`, which in turn calls `InitCPort`.

To dispose of a color graphics port when you are finished using a color window, you normally use the `DisposeWindow` procedure (if you let the Window Manager allocate memory for the window) or the `CloseWindow` procedure (if you allocated memory for the window). You use the `DisposeGWorld` procedure when you are finished with a color graphics port for an offscreen graphics world. These routines automatically call the `CloseCPort` procedure. If you use the `CloseWindow` procedure, you also dispose of the window record containing the graphics port by calling the Memory Manager procedure `DisposePtr`.

OpenCPort

The `OpenCPort` procedure allocates space for and initializes a color graphics port. The Window Manager calls `OpenCPort` for every color window that it creates, and the `NewGWorld` procedure calls `OpenCPort` for every offscreen graphics world that it creates on a Color QuickDraw computer.

```
PROCEDURE OpenCPort (port: CGrafPtr);
```

`port` A pointer to a `CGrafPort` record.

DESCRIPTION

The `OpenCPort` procedure is analogous to `OpenPort` (described in the chapter "Basic QuickDraw"), except that `OpenCPort` opens a `CGrafPort` record instead of a `GrafPort` record. The `OpenCPort` procedure is called by the Window Manager's `NewCWindow` and `GetNewCWindow` procedures, as well as by the Dialog Manager when the appropriate color resources are present. The `OpenCPort` procedure allocates storage for all the structures in the `CGrafPort` record, and then calls `InitCPort` to initialize them. The `InitCPort` procedure does not allocate a color table for the `PixMap` record for the color graphics port; instead, `InitCPort` copies the handle to the current device's CLUT into the `PixMap` record. The initial values for the `CGrafPort` record are shown in Table 4-3.

Table 4-3 Initial values in the `CGrafPort` record

Field	Data type	Initial setting
`device`	`Integer`	0 (the screen)
`portPixMap`	`PixMapHandle`	Handle to the port's `PixMap` record
`portVersion`	`Integer`	$C000
`grafVars`	`Handle`	Handle to a `GrafVars` record where black is assigned to the `rgbOpColor` field, the default highlight color is assigned to the `rgbHiliteColor` field, and all other fields are set to 0
`chExtra`	`Integer`	0
`pnLocHFrac`	`Integer`	The value in this field represents the low word of a `Fixed` number; in decimal, its initial value is 0.5.
`portRect`	`Rect`	`screenBits.bounds` (boundary for entire main screen)
`visRgn`	`RgnHandle`	Handle to a rectangular region coincident with `screenBits.bounds`

Table 4-3 Initial values in the `CGrafPort` record (continued)

Field	Data type	Initial setting
`clipRgn`	`RgnHandle`	Handle to the rectangular region (–32768,–32768,32767,32767)
`bkPixPat`	`Pattern`	White
`rgbFgColor`	`RGBColor`	Black
`rgbBkColor`	`RGBColor`	White
`pnLoc`	`Point`	(0,0)
`pnSize`	`Point`	(1,1)
`pnMode`	`Integer`	`patCopy`
`pnPixPat`	`PixPatHandle`	Black
`fillPixPat`	`PixPatHandle`	Black
`pnVis`	`Integer`	0 (visible)
`txFont`	`Integer`	0 (system font)
`txFace`	`Style`	Plain
`txMode`	`Integer`	`srcOr`
`txSize`	`Integer`	0 (system font size)
`spExtra`	`Fixed`	0
`fgColor`	`LongInt`	`blackColor`
`bkColor`	`LongInt`	`whiteColor`
`colrBit`	`Integer`	0
`patStretch`	`Integer`	0
`picSave`	`Handle`	`NIL`
`rgnSave`	`Handle`	`NIL`
`polySave`	`Handle`	`NIL`
`grafProcs`	`CQDProcsPtr`	`NIL`

The additional structures allocated are the `portPixMap`, `pnPixPat`, `fillPixPat`, `bkPixPat`, and `grafVars` handles, as well as the fields of the `GrafVars` record.

SPECIAL CONSIDERATIONS

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

InitCPort

The `OpenCPort` procedure uses the `InitCPort` procedure to initialize a color graphics port.

```
PROCEDURE InitCPort (port: CGrafPtr);
```

`port` A pointer to a `CGrafPort` record.

DESCRIPTION

The `InitCPort` procedure is analogous to `InitPort` (described in the chapter "Basic QuickDraw"), except `InitCPort` initializes a `CGrafPort` record instead of a `GrafPort` record. The `InitCPort` procedure does not allocate any storage; it merely initializes all the fields in the `CGrafPort` and `GrafVars` records to the default values shown in Table 4-3 on page 4-64.

The `PixMap` record for the new color graphics port is set to be the same as the current device's `PixMap` record. This allows you to create an offscreen graphics world that is identical to the screen's port for drawing offscreen. If you want to use a different set of colors for offscreen drawing, you should create a new `GDevice` record and set it as the current `GDevice` record before opening the `CGrafPort` record.

Remember that `InitCPort` does not copy the data from the current device's CLUT to the color table for the graphics port's `PixMap` record. It simply replaces whatever is in the `PixMap` record's `pmTable` field with a copy of the handle to the current device's CLUT.

If you try to initialize a `GrafPort` record using `InitCPort`, it simply returns without doing anything.

SPECIAL CONSIDERATIONS

The `InitCPort` 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 "Graphics Devices" in this book describes `GDevice` records; the chapter "Offscreen Graphics Worlds" in this book describes how to use offscreen graphics worlds.

CloseCPort

The `CloseCPort` procedure closes a color graphics port. The Window Manager calls this procedure when you close or dispose of a window, and the `DisposeGWorld` procedure calls it when you dispose of an offscreen graphics world containing a color graphics port.

```
PROCEDURE CloseCPort (port: CGrafPtr);
```

`port` A pointer to a `CGrafPort` record.

DESCRIPTION

The `CloseCPort` procedure releases the memory allocated to the `CGrafPort` record. It disposes of the `visRgn`, `clipRgn`, `bkPixPat`, `pnPixPat`, `fillPixPat`, and `grafVars` handles. It also disposes of the graphics port's pixel map, but it doesn't dispose of the pixel map's color table (which is really owned by the `GDevice` record). If you have placed your own color table into the pixel map, either dispose of it before calling `CloseCPort` or store another reference.

SPECIAL CONSIDERATIONS

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

Managing a Color Graphics Pen

You can use the `PenPixPat` procedure to give the graphics pen a pixel pattern so that it draws with a colored, patterned "ink." The QuickDraw painting procedures (such as `PaintRect`) also use this pixel pattern when drawing a shape.

PenPixPat

To set the pixel pattern to be used by the graphics pen in the current color graphics port, use the `PenPixPat` procedure. To assign a pixel pattern as the background pattern, you can use the `BackPixPat` procedure; this allows the `ScrollRect` procedure and the shape-erasing procedures (for example, `EraseRect`) to fill the background with a colored, patterned "ink."

```
PROCEDURE PenPixPat (ppat: PixPatHandle);
```

`ppat` A handle to the pixel pattern to use as the pen pattern.

DESCRIPTION

The `PenPixPat` procedure sets the graphics pen to use the pixel pattern that you specify in the `ppat` parameter. The `PenPixPat` procedure is similar to the basic QuickDraw procedure `PenPat`, except that you pass `PenPixPat` a handle to a multicolored pixel pattern rather than a bit pattern.

The `PenPixPat` procedure stores the handle to the pixel pattern in the `pnPixPat` field of the `CGrafPort` record. Because the handle to the pixel pattern is stored in the `CGrafPort` record, you should not dispose of this handle. QuickDraw removes all references to your pattern from an existing graphics port when you dispose of it.

If you use `PenPixPat` to set a pixel pattern in a basic graphics port, the data in the `pat1Data` field of the `PixPat` record is placed into the `pnPat` field of the `GrafPort` record.

SPECIAL CONSIDERATIONS

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

SEE ALSO

The `PixPat` record is described on page 4-58. To define your own pixel pattern, you can create a pixel pattern resource, which is described on page 4-103, or you can use the `NewPixPat` function, which is described on page 4-88.

The `GrafPort` record is described in the chapter "Basic QuickDraw." To set the graphics pen to use a bit pattern, you can also use the `PenPat` procedure, which is described in the chapter "QuickDraw Drawing" in this book. 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.

Changing the Background Pixel Pattern

Each graphics port has a background pattern that's used when an area is erased (such as by using the `EraseRect` procedure, described in the chapter "QuickDraw Drawing") 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 `bkPixPat` field of every `CGrafPort` record. You can use the `BackPixPat` procedure to change the pixel pattern used as the background color by the current color graphics port.

BackPixPat

To assign a pixel pattern as the background pattern, you can use the `BackPixPat` procedure; this allows the `ScrollRect` procedure and the shape-erasing procedures (for example, `EraseRect`) to fill the background with a colored, patterned "ink."

```
PROCEDURE BackPixPat (ppat: PixPatHandle);
```

`ppat` A handle to the pixel pattern to use as the background pattern.

DESCRIPTION

The `BackPixPat` procedure sets the background pattern for the current graphics device to a pixel pattern. The `BackPixPat` procedure is similar to the basic QuickDraw procedure `BackPat`, except that you pass `BackPixPat` a handle to a multicolored pixel pattern instead of a bit pattern.

The `BackPixPat` procedure stores the handle to the pixel pattern in the `bkPixPat` field of the `CGrafPort` record. Because the handle to the pixel pattern is stored in the `CGrafPort` record, you should not dispose of this handle. QuickDraw removes all references to your pattern from an existing graphics port when you dispose of it.

If you use `BackPixPat` to set a background pixel pattern in a basic graphics port, the data in the `pat1Data` field of the `PixPat` record is placed into the `bkPat` field of the `GrafPort` record.

SPECIAL CONSIDERATIONS

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

SEE ALSO

The `PixPat` record is described on page 4-58. To define your own pixel pattern, you can create a pixel pattern resource, which is described on page 4-103, or you can use the `NewPixPat` function, which is described on page 4-88.

The `GrafPort` record is described in the chapter "Basic QuickDraw." To set the background pattern to a bit pattern, you can also use the `BackPat` procedure, which is described in the chapter "QuickDraw Drawing" in this book. 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 `BackPat` to set the pattern.

Drawing With Color QuickDraw Colors

You can set the foreground and background colors using either Color QuickDraw or Palette Manager routines. If your application uses the Palette Manager, it should set the foreground and background colors with the `PmForeColor` and `PmBackColor` routines, as described in the chapter "Palette Manager" in *Inside Macintosh: Advanced Color Imaging*. Otherwise, it can use the `RGBForeColor` procedure to set the foreground color, and it can use the `RGBBackColor` procedure to set the background color. Both of these Color QuickDraw procedures also operate for basic graphics ports created in System 7. (To set the foreground and background colors for basic graphics ports on older versions of system software, use the `ForeColor` and `BackColor` procedures, described in the chapter "QuickDraw Drawing" in this book.)

To give the graphics pen a pixel pattern so that it draws with a colored, patterned "ink," use the `PenPixPat` procedure. To assign a pixel pattern as the background pattern, you can use the `BackPixPat` procedure; this allows the `ScrollRect` procedure and the shape-erasing procedures (for example, `EraseRect`) to fill the background with the pixel pattern.

To set the color of an individual pixel, use the `SetCPixel` procedure.

The `FillCRect`, `FillCRoundRect`, `FillCOval`, `FillCArc`, `FillCPoly`, and `FillCRgn` procedures allow you to fill shapes with multicolored patterns.

To change the highlight color for the current color graphics port, use the `HiliteColor` procedure. To set values used by arithmetic transfer modes, use the `OpColor` procedure.

As described in "Copying Pixels Between Color Graphics Ports" beginning on page 4-26, you can also use the basic QuickDraw procedures `CopyBits`, `CopyMask`, and `CopyDeepMask` to transfer images between color graphics ports. See the chapter "QuickDraw Drawing" in this book for complete descriptions of these procedures.

RGBForeColor

To change the color of the "ink" used for framing and painting, you can use the `RGBForeColor` procedure.

```
PROCEDURE RGBForeColor (color: RGBColor);
```

`color` An `RGBColor` record.

DESCRIPTION

The `RGBForeColor` procedure lets you set the foreground color to any color available on the current graphics device.

If the current port is defined by a `CGrafPort` record, Color QuickDraw supplies its `rgbFgColor` field with the RGB value that you specify in the `color` parameter, and places the pixel value most closely matching that color in the `fgColor` field. For

indexed devices, the pixel value is an index to the current device's CLUT; for direct devices, the value is the 16-bit or 32-bit equivalent to the RGB value.

If the current port is defined by a `GrafPort` record, basic QuickDraw supplies its `fgColor` field with a color value determined by taking the high bit of each of the red, green, and blue components of the color that you supply in the `color` parameter. Basic QuickDraw uses that 3-bit number to select a color from its eight-color system. Table 4-4 lists the default set of eight colors represented by the global variable `QDColors` (adjusted to match the colors produced on the ImageWriter II printer.)

Table 4-4 The colors defined by the global variable `QDColors`

Value	Color	Red	Green	Blue
0	Black	$0000	$0000	$0000
1	Yellow	$FC00	$F37D	$052F
2	Magenta	$F2D7	$0856	$84EC
3	Red	$DD6B	$08C2	$06A2
4	Cyan	$0241	$AB54	$EAFF
5	Green	$0000	$64AF	$11B0
6	Blue	$0000	$0000	$D400
7	White	$FFFF	$FFFF	$FFFF

SPECIAL CONSIDERATIONS

Color QuickDraw ignores the foreground color (and the background color) when your application draws with a pixel pattern. You can draw with a pixel pattern by using the `PenPixPat` procedure to assign a pixel pattern to the foreground pattern used by the graphics pen; by using the `BackPixPat` procedure to assign a pixel pattern as the background pattern for the current color graphics port; and by using the `FillCRect`, `FillCOval`, `FillCRoundRect`, `FillCArc`, `FillCRgn`, and `FillCPoly` procedures to fill shapes with a pixel pattern.

The `RGBForeColor` procedure is available for basic QuickDraw only in System 7.

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

SEE ALSO

If you want to use one of the eight predefined colors of basic QuickDraw, you can also use the `ForeColor` procedure. The `ForeColor` procedure and the eight-color system of basic QuickDraw are described in the chapter "QuickDraw Drawing" in this book.

To determine the current foreground color, use the `GetForeColor` procedure, which is described on page 4-79.

RGBBackColor

For the current graphics port, you can use the `RGBBackColor` procedure to change the background color (that is, the color of the pixels in the pixel map or bitmap where no drawing has taken place).

```
PROCEDURE RGBBackColor (color: RGBColor);
```

`color` An `RGBColor` record.

DESCRIPTION

The `RGBBackColor` procedure lets you set the background color to any color available on the current graphics device.

If the current port is defined by a `CGrafPort` record, Color QuickDraw supplies its `rgbBkColor` field with the RGB value that you specify in the `color` parameter, and places the pixel value most closely matching that color in the `bkColor` field. For indexed devices, the pixel value is an index to the current device's CLUT; for direct devices, the value is the 16-bit or 32-bit equivalent to the RGB value.

If the current port is defined by a `GrafPort` record, basic QuickDraw supplies its `fgColor` field with a color value determined by taking the high bit of each of the red, green, and blue components of the color that you supply in the `color` parameter. Basic QuickDraw uses that 3-bit number to select a color from its eight-color system. Table 4-4 on page 4-71 lists the default colors.

SPECIAL CONSIDERATIONS

Because a pixel pattern already contains color, Color QuickDraw ignores the background color (and the foreground color) when your application draws with a pixel pattern. You can draw with a pixel pattern by using the `PenPixPat` procedure to assign a pixel pattern to the foreground pattern used by the graphics pen; by using the `BackPixPat` procedure to assign a pixel pattern as the background pattern for the current color graphics port; and by using the `FillCRect`, `FillCOval`, `FillCRoundRect`, `FillCArc`, `FillCRgn`, and `FillCPoly` procedures to fill shapes with a pixel pattern.

This procedure is available for basic QuickDraw only in System 7.

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

SEE ALSO

If you want to use one of the eight predefined colors of basic QuickDraw, you can also use the `BackColor` procedure. The `BackColor` procedure and the eight-color system of basic QuickDraw are described in the chapter "QuickDraw Drawing" in this book.

To determine the current background color, use the `GetBackColor` procedure, which is described on page 4-80.

SetCPixel

To set the color of an individual pixel, use the `SetCPixel` procedure.

```
PROCEDURE SetCPixel (h,v: Integer; cPix: RGBColor);
```

`h`	The horizontal coordinate of the point at the upper-left corner of the pixel.
`v`	The vertical coordinate of the point at the upper-left corner of the pixel.
`cPix`	An `RGBColor` record.

DESCRIPTION

For the pixel at the location you specify in the `h` and `v` parameters, the `SetCPixel` procedure sets a pixel value that most closely matches the RGB color that you specify in the `cPix` parameter. On an indexed color system, the `SetCPixel` procedure sets the pixel value to the index of the best-matching color in the current device's CLUT. In a direct environment, the `SetCPixel` procedure sets the pixel value to a 16-bit or 32-bit direct pixel value.

SPECIAL CONSIDERATIONS

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

SEE ALSO

To determine the color of an individual pixel, use the `GetCPixel` procedure, which is described on page 4-80.

FillCRect

Use the `FillCRect` procedure to fill a rectangle with a pixel pattern.

```
PROCEDURE FillCRect (r: Rect; ppat: PixPatHandle);
```

`r`	The rectangle to be filled.
`ppat`	A handle to the `PixPat` record for the pixel pattern to be used for the fill.

DESCRIPTION

Using the `patCopy` pattern mode, the `FillCRect` procedure fills the rectangle you specify in the `r` parameter with the pixel pattern defined by a `PixPat` record, the handle for which you pass in the `ppat` parameter. This procedure ignores the `pnPat`, `pnMode`, and `bkPat` fields of the current graphics port and leaves the pen location unchanged.

SPECIAL CONSIDERATIONS

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

FillCRoundRect

Use the `FillCRoundRect` procedure to fill a rounded rectangle with a pixel pattern.

```
PROCEDURE FillCRoundRect (r: Rect; ovalWidth,ovalHeight: Integer;
                          ppat: PixPatHandle);
```

`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.
`ppat`	A handle to the `PixPat` record for the pixel pattern to be used for the fill.

DESCRIPTION

Using the `patCopy` pattern mode, the `FillCRoundRect` procedure fills the rectangle you specify in the `r` parameter with the pixel pattern defined in a `PixPat` record, the handle for which you pass in the `ppat` parameter. Use the `ovalWidth` and `ovalHeight` parameters to specify the diameters of curvature for the corners. This procedure ignores the `pnPat`, `pnMode`, and `bkPat` fields of the current graphics port and leaves the pen location unchanged.

SPECIAL CONSIDERATIONS

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

FillCOval

Use the `FillCOval` procedure to fill an oval with a pixel pattern.

```
PROCEDURE FillCOval (r: Rect; ppat: PixPatHandle);
```

`r`	The rectangle containing the oval to be filled.
`ppat`	A handle to the `PixPat` record for the pixel pattern to be used for the fill.

DESCRIPTION

Using the `patCopy` pattern mode and the pixel pattern defined in the `PixPat` record (the handle for which you pass in the `ppat` parameter), the `FillCOval` procedure fills an oval just inside the bounding rectangle that you specify in the `r` parameter. This procedure ignores the `pnPat`, `pnMode`, and `bkPat` fields of the current graphics port and leaves the pen location unchanged.

SPECIAL CONSIDERATIONS

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

FillCArc

Use the `FillCArc` procedure to fill a wedge with a pixel pattern.

```
PROCEDURE FillCArc (r: Rect; startAngle,arcAngle: Integer;
                    ppat: PixPatHandle);
```

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

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

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

`ppat` A handle to the `PixPat` record for the pixel pattern to be used for the fill.

DESCRIPTION

Using the `patCopy` pattern mode and the pixel pattern defined in a `PixPat` record (the handle for which you pass in the `ppat` parameter), the `FillCArc` procedure fills a wedge of the oval bounded by the rectangle that you specify in the `r` parameter. As in the `FrameArc` procedure, described in the chapter "QuickDraw Drawing" in this book, use the `startAngle` and `arcAngle` parameters to define the arc of the wedge. This procedure ignores the `pnPat`, `pnMode`, and `bkPat` fields of the current graphics port and leaves the pen location unchanged.

SPECIAL CONSIDERATIONS

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

FillCPoly

Use the `FillCPoly` procedure to fill a polygon with a pixel pattern.

```
PROCEDURE FillCPoly (poly: PolyHandle; ppat: PixPatHandle);
```

`poly` A handle to the polygon to be filled.

`ppat` A handle to the `PixPat` record for the pixel pattern to be used for the fill.

DESCRIPTION

Using the `patCopy` pattern mode and the pixel pattern defined in a `PixPat` record (the handle for which you pass in the `ppat` parameter), the `FillCPoly` procedure fills the polygon whose handle you pass in the `poly` parameter. This procedure ignores the `pnPat`, `pnMode`, and `bkPat` fields of the current graphics port and leaves the pen location unchanged.

SPECIAL CONSIDERATIONS

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

FillCRgn

Use the `FillCRgn` procedure to fill a region with a pixel pattern.

```
PROCEDURE FillCRgn (rgn: RgnHandle; ppat: PixPatHandle);
```

`rgn`	A handle to the region to be filled.
`ppat`	A handle to the `PixPat` record for the pixel pattern to be used for the fill.

DESCRIPTION

Using the `patCopy` pattern mode and the pixel pattern defined in a `PixPat` record (the handle for which you pass in the `ppat` parameter), the `FillCRgn` procedure fills the region whose handle you pass in the `rgn` parameter. This procedure ignores the `pnPat`, `pnMode`, and `bkPat` fields of the current graphics port and leaves the pen location unchanged.

SPECIAL CONSIDERATIONS

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

OpColor

Use the `OpColor` procedure to set the maximum color values for the `addPin` and `subPin` arithmetic transfer modes, and the weight color for the `blend` arithmetic transfer mode.

```
PROCEDURE OpColor (color: RGBColor);
```

`color` An `RGBColor` record that defines a color.

DESCRIPTION

If the current port is defined by a `CGrafPort` record, the `OpColor` procedure sets the red, green, and blue values used by the `addPin`, `subPin`, and `blend` arithmetic transfer modes. You specify these red, green, and blue values in the `RGBColor` record, and you specify this record in the `color` parameter. This information is actually stored in the `rgbOpColor` field of the `GrafVars` record, but you should never need to refer to it directly.

If the current graphics port is defined by a `GrafPort` record, `OpColor` has no effect.

SEE ALSO

Arithmetic transfer modes are described in "Arithmetic Transfer Modes" beginning on page 4-38.

HiliteColor

Use the `HiliteColor` procedure to change the highlight color for the current color graphics port.

```
PROCEDURE HiliteColor (color: RGBColor);
```

`color` An `RGBColor` record that defines the highlight color.

DESCRIPTION

The `HiliteColor` procedure changes the highlight color for the current color graphics port. All drawing operations that use the `hilite` transfer mode use the highlight color. When a color graphics port is created, its highlight color is initialized from the global variable `HiliteRGB`. (This information is stored in the `rgbHiliteColor` field of the `GrafVars` record, but you should never need to refer to it directly.)

If the current graphics port is a basic graphics port, `HiliteColor` has no effect.

SEE ALSO

The `hilite` mode is described in "Highlighting" beginning on page 4-41.

Determining Current Colors and Best Intermediate Colors

The `GetForeColor` and `GetBackColor` procedures allow you to obtain the foreground and background colors for the current graphics port, both basic and color. You can use the `GetCPixel` procedure to determine the color of an individual pixel. The `GetGray` function can do more than its name implies: it can return the best gray for a given graphics device, but it can also return the best available intermediate color between any two colors.

GetForeColor

Use the `GetForeColor` procedure to obtain the color of the foreground color for the current graphics port.

```
PROCEDURE GetForeColor (VAR color: RGBColor);
```

`color` An `RGBColor` record.

DESCRIPTION

In the `color` parameter, the `GetForeColor` procedure returns the `RGBColor` record for the foreground color of the current graphics port. This procedure operates for graphics ports defined by both the `GrafPort` and `CGrafPort` records. If the current graphics port is defined by a `CGrafPort` record, the returned value is taken directly from the `rgbFgColor` field.

If the current graphics port is defined by a `GrafPort` record, then only eight possible RGB values can be returned. These eight values are determined by the values in a global variable named `QDColors`, which is a handle to a color table containing the current QuickDraw colors. These colors are listed in Table 4-4 on page 4-71. This default set of colors has been adjusted to match the colors produced on the ImageWriter II printer.

SPECIAL CONSIDERATIONS

This procedure is available for basic QuickDraw only in System 7.

SEE ALSO

You can use the `RGBForeColor` procedure, described on page 4-70, to change the foreground color.

GetBackColor

Use the `GetBackColor` procedure to obtain the background color of the current graphics port.

```
PROCEDURE GetBackColor (VAR color: RGBColor);
```

`color` An `RGBColor` record.

DESCRIPTION

In the `color` parameter, the `GetBackColor` procedure returns the `RGBColor` record for the background color of the current graphics port. This procedure operates for graphics ports defined by both the `GrafPort` and `CGrafPort` records. If the current graphics port is defined by a `CGrafPort` record, the returned value is taken directly from the `rgbBkColor` field.

If the current graphics port is defined by a `GrafPort` record, then only eight possible colors can be returned. These eight colors are determined by the values in a global variable named `QDColors`, which is a handle to a color table containing the current QuickDraw colors. These colors are listed in Table 4-4 on page 4-71.

SPECIAL CONSIDERATIONS

This procedure is available for basic QuickDraw only in System 7.

SEE ALSO

You can use the `RGBBackColor` procedure, described on page 4-72, to change the background color.

GetCPixel

To determine the color of an individual pixel, use the `GetCPixel` procedure.

```
PROCEDURE GetCPixel (h,v: Integer; VAR cPix: RGBColor);
```

`h` The horizontal coordinate of the point at the upper-left corner of the pixel.

`v` The vertical coordinate of the point at the upper-left corner of the pixel.

`cPix` The `RGBColor` record for the pixel.

DESCRIPTION

In the `cPix` parameter, the `GetCPixel` procedure returns the RGB color for the pixel at the location you specify in the `h` and `v` parameters.

SEE ALSO

You can use the `SetCPixel` procedure, described on page 4-73, to change the color of this pixel.

GetGray

To determine the best intermediate color between two colors on a given graphics device, use the `GetGray` function.

```
FUNCTION GetGray (device: GDHandle; backGround: RGBColor;
                  VAR foreGround: RGBColor): Boolean;
```

`device` A handle to the graphics device for which an intermediate color or gray is needed.

`backGround`
The `RGBColor` record for one of the two colors for which you want an intermediate color.

`foreGround`
On input, the `RGBColor` record for the other of the two colors; upon completion, the best intermediate color between these two.

DESCRIPTION

The `GetGray` function determines the midpoint values for the red, green, and blue values of the two colors you specify in the `backGround` and `foreGround` parameters. In the `device` parameter, supply a handle to the graphics device; in the `backGround` and `foreGround` parameters, supply `RGBColor` records for the two colors for which you want the best intermediate RGB color. When `GetGray` completes, it returns the best intermediate color in the `foreGround` parameter.

One use for `GetGray` is to return the best gray. For example, when dimming an object, supply black and white as the two colors, and `GetGray` returns the best available gray that lies between them. (The Menu Manager does this when dimming unavailable menu items.)

If no gray is available (or if no distinguishable third color is available), the `foreGround` parameter is unchanged, and the function returns `FALSE`. If at least one gray or intermediate color is available, it is returned in the `foreGround` parameter, and the function returns `TRUE`.

Calculating Color Fills

Just as basic QuickDraw provides a pair of procedures (`SeedFill` and `CalcMask`) to help you determine the results of filling operations on portions of bitmaps, Color QuickDraw provides the `SeedCFill` and `CalcCMask` procedures to help you determine the results of filling operations on portions of pixel maps.

SeedCFill

To determine how far filling will extend to pixels matching the color of a particular pixel, use the `SeedCFill` procedure.

```
PROCEDURE SeedCFill (srcBits,dstBits: BitMap;
                     srcRect,dstRect: Rect; seedH,seedV: Integer;
                     matchProc: ProcPtr; matchData: LongInt);
```

`srcBits`	The source image. If the image is in a pixel map, you must coerce its `PixMap` record to a `BitMap` record.
`dstBits`	The destination mask.
`srcRect`	The rectangle of the source image.
`dstRect`	The rectangle of the destination image.
`seedH`	The horizontal position of the seed point.
`seedV`	The vertical position of the seed point.
`matchProc`	An optional color search function.
`matchData`	Data for the optional color search function.

DESCRIPTION

The `SeedCFill` procedure generates a mask showing where the pixels in an image can be filled from a starting point, like the paint pouring from the MacPaint paint-bucket tool. The `SeedCFill` procedure returns this mask in the `dstBits` parameter. This mask is a bitmap filled with 1's to indicate all pixels adjacent to a seed point whose colors do not exactly match the `RGBColor` record for the pixel at the seed point. You can then use this mask with the `CopyBits`, `CopyMask`, and `CopyDeepMask` procedures.

You specify a source image in the `srcBits` parameter, and in the `srcRect` parameter you specify a rectangle within that source image. You specify where to begin seeding in the `seedH` and `seedV` parameters, which must be the horizontal and vertical coordinates of a point in the local coordinate system of the source bitmap. By default, the 1's returned in the mask indicate all pixels adjacent to the seed point whose pixel values do not exactly match the pixel value of the pixel at the seed point. To use this default, set the `matchProc` and `matchData` parameters to 0.

In generating the mask, `SeedCFill` uses the `CopyBits` procedure to convert the source image to a 1-bit mask. The `SeedCFill` procedure installs a default color search function that returns 0 if the pixel value matches that of the seed point; all other pixel values return 1's.

The `SeedCFill` procedure does not scale: the source and destination rectangles must be the same size. Calls to `SeedCFill` are not clipped to the current port and are not stored into QuickDraw pictures.

You can customize `SeedCFill` by writing your own color search function and pointing to it in the `matchProc` procedure; `SeedCFill` will then use your procedure instead of the default. You can use the `matchData` parameter for whatever you'd like. In the `matchData` parameter, for instance, your application could pass the handle to a color table. Your color search function could then check whether the pixel value for the pixel currently under analysis matches any of the colors in the table.

SEE ALSO

See "Application-Defined Routine" on page 4-101 for a description of how to customize the `SeedCFill` procedure.

CalcCMask

To determine where filling will not occur when filling from the outside of a rectangle, you can use the `CalcCMask` procedure, which indicates pixels that match, or are surrounded by pixels that match, a particular color.

```
PROCEDURE CalcCMask (srcBits,dstBits: BitMap;
                     srcRect,dstRect: Rect;
                     seedRGB: RGBColor; matchProc: ProcPtr;
                     matchData: LongInt);
```

`srcBits` The source image. If the image is in a pixel map, you must coerce its `PixMap` record to a `BitMap` record.

`dstBits` The destination image, a `BitMap` record.

`srcRect` The rectangle of the source image.

`dstRect` The rectangle of the destination image.

`seedRGB` An `RGBColor` record specifying the color for pixels that should not be filled.

`matchProc` An optional matching procedure.

`matchData` Data for the optional matching procedure.

DESCRIPTION

The `CalcCMask` procedure generates a mask showing where pixels in an image cannot be filled from any of the outer edges of the rectangle you specify. The `CalcCMask` procedure returns this mask in the `dstBits` parameter. This mask is a bitmap filled with 1's only where the pixels in the source image *cannot* be filled. You can then use this mask with the `CopyBits`, `CopyMask`, and `CopyDeepMask` procedures.

You specify a source image in the `srcBits` parameter, and in the `srcRect` parameter you specify a rectangle within that source image. Starting from the edges of this rectangle, `CalcCMask` calculates which pixels *cannot* be filled. By default, `CalcCMask` returns 1's in the mask to indicate which pixels have the exact color that you specify in the `seedRGB` parameter, as well as which pixels are enclosed by shapes whose outlines consist entirely of pixels with this color.

For instance, if the source image in `srcBits` contains a dark blue rectangle on a red background, and your application sets `seedRGB` equal to dark blue, then `CalcCMask` returns a mask with 1's in the positions corresponding to the edges and interior of the rectangle, and 0's outside of the rectangle.

If you set the `matchProc` and `matchData` parameters to 0, `CalcCMask` uses the exact color specified in the `RGBColor` record that you supply in the `seedRGB` parameter. You can customize `CalcCMask` by writing your own color search function and pointing to it in the `matchProc` procedure; your color search function might, for example, search for colors that approximate the color specified in the `RGBColor` record. As with `SeedCFill`, you can then use the `matchData` parameter in any manner useful for your application.

The `CalcCMask` procedure does not scale—the source and destination rectangles must be the same size. Calls to `CalcCMask` are not clipped to the current port and are not stored into QuickDraw pictures.

SEE ALSO

See "Application-Defined Routine" on page 4-101 for a description of how to customize the `CalcCMask` procedure.

Creating, Setting, and Disposing of Pixel Maps

QuickDraw automatically creates pixel maps when you create a color window with the `GetNewCWindow` or `NewCWindow` function (described in the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*), when you create offscreen graphics worlds with the `NewGWorld` function (described in the chapter "Offscreen Graphics Worlds" in this book), and when you use the `OpenCPort` function. QuickDraw also disposes of a pixel map when it disposes of a color graphics port. Although your application typically won't need to create or dispose of pixel maps, you can use the `NewPixMap` function and the `CopyPixMap` procedure to create them, and you can use the `DisposePixMap` procedure to dispose of them. Although you should never need to do so, you can also set the pixel map for the current color graphics port by using the `SetPortPix` procedure.

NewPixMap

Although you typically don't need to call this routine in your application code, you can use the `NewPixMap` function to create a new, initialized `PixMap` record.

```
FUNCTION NewPixMap: PixMapHandle;
```

DESCRIPTION

The `NewPixMap` function creates a new, initialized `PixMap` record and returns a handle to it. All fields of the `PixMap` record are copied from the current device's `PixMap` record except the color table. In System 7, the `hRes` and `vRes` fields are set to 72 dpi, no matter what values the current device's `PixMap` record contains. A handle to the color table is allocated but not initialized.

You typically don't need to call this routine. `PixMap` records are created for you when you create a window using the Window Manager functions `NewCWindow` and `GetNewCWindow` and when you create an offscreen graphics world with the `NewGWorld` function.

If your application creates a pixel map, your application must initialize the `PixMap` record's color table to describe the pixels. You can use the `GetCTable` function (described on page 4-92) to read such a table from a resource file; you can then use the `DisposeCTable` procedure (described on page 4-93) to dispose of the `PixMap` record's color table and replace it with the one returned by `GetCTable`.

SPECIAL CONSIDERATIONS

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

CopyPixMap

Although you typically don't need to call this routine in your application code, you can use the `CopyPixMap` procedure to duplicate a `PixMap` record.

```
PROCEDURE CopyPixMap (srcPM,dstPM: PixMapHandle);
```

`srcPM` A handle to the `PixMap` record to be copied.

`dstPM` A handle to the duplicated `PixMap` record.

DESCRIPTION

The `CopyPixMap` procedure copies the contents of the source `PixMap` record to the destination `PixMap` record. The contents of the color table are copied, so the destination `PixMap` has its own copy of the color table. Because the `baseAddr` field of the `PixMap` record is a pointer, the pointer, but not the image itself, is copied.

SetPortPix

Although you should never need to do so, you can set the pixel map for the current color graphics port by using the `SetPortPix` procedure.

```
PROCEDURE SetPortPix (pm: PixMapHandle);
```

`pm` A handle to the `PixMap` record.

DESCRIPTION

The `SetPortPix` procedure replaces the `portPixMap` field of the current `CGrafPort` record with the handle you specify in the `pm` parameter.

SPECIAL CONSIDERATIONS

The `SetPortPix` procedure is analogous to the basic QuickDraw procedure `SetPortBits`, which sets the bitmap for the current basic graphics port. The `SetPortPix` procedure has no effect when used with a basic graphics port. Similarly, `SetPortBits` has no effect when used with a color graphics port.

Both `SetPortPix` and `SetPortBits` allow you to perform drawing and calculations on a buffer other than the screen. However, instead of using these procedures, you should use the offscreen graphics capabilities described in the chapter "Offscreen Graphics Worlds."

DisposePixMap

Although you typically don't need to call this routine in your application code, you can use the `DisposePixMap` procedure to dispose of a `PixMap` record. The `DisposePixMap` procedure is also available as the `DisposPixMap` procedure.

```
PROCEDURE DisposePixMap (pm: PixMapHandle);
```

`pm` A handle to the `PixMap` record to be disposed of.

DESCRIPTION

The `DisposePixMap` procedure disposes of the `PixMap` record and its color table. The `CloseCPort` procedure calls `DisposePixMap`.

SPECIAL CONSIDERATIONS

If your application uses `DisposePixMap`, take care that it does not dispose of a `PixMap` record whose color table is the same as the current device's CLUT.

Creating and Disposing of Pixel Patterns

Pixel patterns can use colors at any pixel depth and can be of any width and height that's a power of 2. To create a pixel pattern, you typically define it in a `'ppat'` resource, which you store in a resource file. To retrieve the pixel pattern stored in a `'ppat'` resource, you can use the `GetPixPat` function.

Color QuickDraw also allows you to create and dispose of pixel patterns by using the `NewPixPat`, `CopyPixPat`, `MakeRGBPat`, and `DisposePixPat` routines, although generally you should create them in `'ppat'` resources (described on page 4-103).

When your application is finished using a pixel pattern, it should dispose of it with the `DisposePixPat` procedure.

GetPixPat

To get a pixel pattern (`'ppat'`) resource stored in a resource file, you can use the `GetPixPat` function.

```
FUNCTION GetPixPat (patID: Integer): PixPatHandle;
```

`patID` The resource ID for a resource of type `'ppat'`.

DESCRIPTION

The `GetPixPat` function returns a handle to the pixel pattern having the resource ID you specify in the `patID` parameter. The `GetPixPat` function calls the following Resource Manager function with these parameters:

```
GetResource('ppat', patID);
```

If a `'ppat'` resource with the ID that you request does not exist, the `GetPixPat` function returns `NIL`.

When you are finished with the pixel pattern, use the `DisposePixPat` procedure (described on page 4-91).

SPECIAL CONSIDERATIONS

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

SEE ALSO

The `'ppat'` resource format is described on page 4-103. See the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information about resources, the Resource Manager, and the `GetResource` function.

NewPixPat

Although you should generally create a pixel pattern in a `'ppat'` resource and retrieve it with the `GetPixPat` function, you can use the `NewPixPat` function to create a new pixel pattern.

```
FUNCTION NewPixPat: PixPatHandle;
```

DESCRIPTION

The `NewPixPat` function creates a new `PixPat` record (described on page 4-58) and returns a handle to it. This function calls the `NewPixMap` function to allocate the pattern's `PixMap` record (described on page 4-46) and initialize it to the same settings as the pixel map of the current `GDevice` record—that is, as stored in the `gdPMap` field of the global variable `TheGDevice`. This function also sets the `pat1Data` field of the new `PixPat` record to a 50 percent gray pattern. `NewPixPat` allocates new handles for the `PixPat` record's data, expanded data, expanded map, and color table but does not initialize them; instead, your application must initialize them.

Set the `rowBytes`, `bounds`, and `pixelSize` fields of the pattern's `PixMap` record to the dimensions of the desired pattern. The `rowBytes` value should be equal to

(width of bounds) × `pixelSize` / 8

The `rowBytes` value need not be even. The width and height of the bounds must be a power of 2. Each scan line of the pattern must be at least 1 byte in length—that is, ([width of bounds] × `pixelSize`) must be at least 8.

Your application can explicitly specify the color corresponding to each pixel value with a color table. The color table for the pattern must be placed in the `pmTable` field in the pattern's `PixMap` record.

Including the `PixPat` record itself, `NewPixPat` allocates a total of five handles. The sizes of the handles to the `PixPat` and `PixMap` records are the sizes of their respective data structures. The other three handles are initially small in size. Once the pattern is drawn, the size of the expanded data is proportional to the size of the pattern data, but adjusted to the depth of the screen. The color table size is the size of the record plus 8 bytes times the number of colors in the table.

When you are finished using the pixel pattern, use the `DisposePixPat` procedure, which is described on page 4-91, to make the memory used by the pixel pattern available again.

SPECIAL CONSIDERATIONS

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

CopyPixPat

Use the `CopyPixPat` procedure to copy the contents of one pixel pattern to another.

```
PROCEDURE CopyPixPat (srcPP,dstPP: PixPatHandle);
```

`srcPP` A handle to a source pixel pattern, the contents of which you want to copy.

`dstPP` A handle to a destination pixel pattern, into which you want to copy the contents of the pixel pattern in the `srcPP` parameter.

DESCRIPTION

The `CopyPixPat` procedure copies the contents of one `PixPat` record (the handle to which you pass in the `srcPP` parameter) into another `PixPat` record (the handle to which you pass in the `dstPP` parameter). The `CopyPixPat` procedure copies all of the fields in the source `PixPat` record, including the contents of the data handle, expanded data handle, expanded map, pixel map handle, and color table.

SEE ALSO

The `PixPat` record is described on page 4-58.

MakeRGBPat

To create the appearance of otherwise unavailable colors on indexed devices, you can use the `MakeRGBPat` procedure.

```
PROCEDURE MakeRGBPat (ppat: PixPatHandle; myColor: RGBColor);
```

`ppat` A handle to hold the generated pixel pattern.

`myColor` An `RGBColor` record that defines the color you want to approximate.

DESCRIPTION

The `MakeRGBPat` procedure generates a `PixPat` record that, when used to draw a pixel pattern, approximates the color you specify in the `myColor` parameter. For example, if your application draws to an indexed device that supports 4 bits per pixel, you only have 16 colors available if you simply set the foreground color and draw. If you use `MakeRGBPat` to create a pattern, and then draw using that pattern, you effectively get 125 different colors. If the graphics device has 8 bits per pixel, you effectively get 2197 colors. (More colors are theoretically possible; this implementation opted for a fast pattern selection rather than the best possible pattern selection.)

For a pixel pattern, the `patMap^^.bounds` field of the `PixPat` record always contains the values (0,0,8,8), and the `patMap^^.rowbytes` field equals 2.

SPECIAL CONSIDERATIONS

Because patterns produced with `MakeRGBPat` aren't usually solid—they provide a selection of colors by alternating between colors, with up to four colors in a pattern—lines that are only one pixel wide may not look good.

When `MakeRGBPat` creates a `ColorTable` record, it fills in only the `rgb` fields of its `ColorSpec` records; the `value` fields are computed at the time the drawing actually takes place, using the current pixel depth for the system.

DisposePixPat

Use the `DisposePixPat` procedure to release the storage allocated to a pixel pattern. The `DisposePixPat` procedure is also available as the `DisposPixPat` procedure.

```
PROCEDURE DisposePixPat (ppat: PixPatHandle);
```

`ppat` A handle to the pixel pattern to be disposed of.

DESCRIPTION

The `DisposePixPat` procedure disposes of the data handle, expanded data handle, and pixel map handle allocated to the pixel pattern that you specify in the `ppat` parameter.

Creating and Disposing of Color Tables

You use a color table, which is defined by a data structure of type `ColorTable`, to specify colors in the form of `RGBColor` records. You can create and store color tables in `'clut'` resources. To retrieve a color table stored in a `'clut'` resource, you can use the `GetCTable` function. To dispose of the handle allocated for a color table, you use the `DisposeCTable` procedure.

The Palette Manager, described in the chapter "Palette Manager" in *Inside Macintosh: Advanced Color Imaging,* has additional routines that enable you to copy colors between palettes and color tables and to restore the default colors to a CLUT belonging to a graphics device.

The Color Manager, described in the chapter "Color Manager" in *Inside Macintosh: Advanced Color Imaging,* contains low-level routines for directly manipulating the fields of the CLUT on a graphics device; most applications do not need to use those routines.

GetCTable

To get a color table stored in a `'clut'` resource, use the `GetCTable` function.

```
FUNCTION GetCTable (ctID: Integer): CTabHandle;
```

`ctID` The resource ID of a `'clut'` resource.

DESCRIPTION

For the color table defined in the `'clut'` resource that you specify in the `ctID` parameter, the `GetCTable` function returns a handle to a `ColorTable` record. If the `'clut'` resource with that ID is not found, `GetCTable` returns `NIL`.

If you place this handle in the `pmTable` field of a `PixMap` record, you should first use the `DisposeCTable` procedure to dispose of the handle already there.

If you modify a `ColorTable` record, you should invalidate it by changing its `ctSeed` field. An easy way to do this is with the `CTabChanged` procedure, described on page 4-97.

The `GetCTable` function recognizes a number of standard `'clut'` resource IDs. You can obtain the default grayscale color table for a given pixel depth by calling `GetCTable`, adding 32 (decimal) to the pixel depth, and passing this value in the `ctID` parameter, as shown in Table 4-5.

Table 4-5 The default color tables for grayscale graphics devices

Pixel depth	Resource ID	Color table composition
1	33	Black, white
2	34	Black, 33% gray, 66% gray, white
4	36	Black, 14 shades of gray, white
8	40	Black, 254 shades of gray, white

For full color, you can obtain the default color tables by adding 64 to the pixel depth and passing this in the `ctID` parameter, as shown in Table 4-6. These default color tables are illustrated in Plate 1 at the front of this book.

Table 4-6 The default color tables for color graphics devices

Pixel depth	Resource ID	Color table composition
2	66	Black, 50% gray, highlight color, white
4	68	Black, 14 colors including the highlight color, white
8	72	Black, 254 colors including the highlight color, white

SPECIAL CONSIDERATIONS

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

SEE ALSO

The `'clut'` resource is described on page 4-104.

DisposeCTable

Use the `DisposeCTable` procedure to dispose of a `ColorTable` record. The `DisposeCTable` procedure is also available as the `DisposCTable` procedure.

```
PROCEDURE DisposeCTable (cTable: CTabHandle);
```

`cTable` A handle to a `ColorTable` record.

DESCRIPTION

The `DisposeCTable` procedure disposes of the `ColorTable` record whose handle you pass in the `cTable` parameter.

Retrieving Color QuickDraw Result Codes

Most QuickDraw routines do not return result codes. However, you can use the `QDError` function to get a result code from the last applicable Color QuickDraw or Color Manager routine that you called.

QDError

To get a result code from the last applicable Color QuickDraw or Color Manager routine that you called, use the `QDError` function.

```
FUNCTION QDError: Integer;
```

DESCRIPTION

The `QDError` function returns the error result from the last applicable Color QuickDraw or Color Manager routine. On a system with only basic QuickDraw, `QDError` always returns `noErr`.

The `QDError` function is helpful in determining whether insufficient memory caused a drawing operation—particularly those involving regions, polygons, pictures, and images copied with `CopyBits`—to fail in Color QuickDraw.

Basic QuickDraw uses stack space for work buffers. For complex operations such as depth conversion, dithering, and image resizing, stack space may not be sufficient. Color QuickDraw attempts to get temporary memory from other parts of the system. If that is still not enough, `QDError` returns the `nsStackErr` error. If your application receives this result, reduce the memory required by the operation—for example, divide the image into left and right halves—and try again.

When you record drawing operations in an open region, the resulting region description may overflow the 64 KB limit. Should this happen, `QDError` returns `regionTooBigError`. Since the resulting region is potentially corrupt, the `CloseRgn` procedure (described in the chapter "QuickDraw Drawing" in this book) returns an empty region if it detects `QDError` has returned `regionTooBigError`. A similar error, `rgnTooBigErr`, can occur when using the `BitMapToRegion` function (described in the chapter "Basic QuickDraw" in this book) to convert a bitmap to a region.

The `BitMapToRegion` function can also generate the `pixmapTooDeepErr` error if a `PixMap` record is supplied that is greater than 1 bit per pixel. You may be able to recover from this problem by coercing your `PixMap` record into a 1-bit `PixMap` record and calling the `BitMapToRegion` function again.

RESULT CODES

noErr	0	No error
paramErr	–50	Illegal parameter to NewGWorld
	–143	CopyBits couldn't allocate required temporary memory
	–144	Ran out of stack space while drawing polygon
noMemForPictPlaybackErr	–145	Insufficient memory for drawing the picture
regionTooBigError	–147	Region too big or complex
pixmapTooDeepErr	–148	Pixel map is deeper than 1 bit per pixel
nsStackErr	–149	Insufficient stack
cMatchErr	–150	Color2Index failed to find an index
cTempMemErr	–151	Failed to allocate memory for temporary structures
cNoMemErr	–152	Failed to allocate memory for structure
cRangeErr	–153	Range error on color table request
cProtectErr	–154	ColorTable record entry protection violation
cDevErr	–155	Invalid type of graphics device
cResErr	–156	Invalid resolution for MakeITable
cDepthErr	–157	Invalid pixel depth specified to NewGWorld
rgnTooBigErr	–500	Bitmap would convert to a region greater than 64 KB

In addition to these result codes, QDErr also returns result codes from the Memory Manager.

SECIAL CONSIDERATIONS

The QDError function does not report errors returned by basic QuickDraw.

SEE ALSO

Listing 3-8 on page 3-28, Listing 3-10 on page 3-30, and Listing 3-11 on page 3-33 in the chapter "QuickDraw Drawing" in this book—and Listing 7-8 on page 7-20 in the chapter "Pictures" in this book—illustrate how to use QDError to report insufficient memory conditions for various drawing operations.

The NewGWorld function is described in the chapter "Offscreen Graphics Worlds" in this book. The Color2Index function and the MakeITable procedure are described in the chapter "Color Manager" in *Inside Macintosh: Advanced Color Imaging*. Graphics devices are described in the chapter "Graphics Devices" in this book. Memory Manager result codes are listed in *Inside Macintosh: Memory.*

Customizing Color QuickDraw Operations

For each shape that QuickDraw can draw, there are procedures that perform these basic graphics operations on the shape: framing, painting, erasing, inverting, and filling. As described in the chapter "QuickDraw Drawing" in this book, 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.

The `grafProcs` field of a `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. This record of pointers is defined by a data structure of type `CQDProcs`. 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, QuickDraw provides the `SetStdCProcs` procedure. You can use the `SetStdCProcs` procedure to set all the fields of the `CQDProcs` record to point to the standard routines. You can then reset the ones with which you are concerned.

SetStdCProcs

You can use the `SetStdCProcs` procedure to get a `CQDProcs` record with fields that point to Color QuickDraw's standard low-level routines. You can replace these low-level routines with your own, and then point to your modified `CQDProcs` record in the `grafProcs` field of a `CGrafPort` record to change Color QuickDraw's standard low-level behavior.

```
PROCEDURE SetStdCProcs (VAR cProcs: CQDProcs);
```

`cProcs` Upon completion, a `CQDProcs` record with fields that point to Color QuickDraw's standard low-level routines.

DESCRIPTION

In the `cProcs` parameter, the `SetStdCProcs` procedure returns a `CQDProcs` record with fields that point to the standard low-level routines. You can change one or more fields to point to your own routines and then set the color graphics port to use this modified `CQDProcs` record.

SPECIAL CONSIDERATIONS

When drawing in a color graphics port, your application must always use `SetStdCProcs` instead of `SetStdProcs`.

SEE ALSO

The routines you install in the `CQDProcs` record must have the same calling sequences as the standard basic QuickDraw routines, which are described in the chapter "QuickDraw Drawing" in this book. The `SetStdProcs` procedure is also described in the chapter "QuickDraw Drawing."

The chapter "Pictures" in this book describes how to replace the low-level routines that read and write pictures.

The data structure of type `CQDProcs` is described on page 4-60.

Reporting Data Structure Changes to QuickDraw

In quest of faster execution time, some applications directly modify `ColorTable`, `PixPat`, `GrafPort`, `CGrafPort`, or `GDevice` records rather than using the routines provided for that purpose. Direct manipulation of the fields of these records can cause problems now, and may cause additional problems as QuickDraw continues to evolve.

For example, the Color Manager (described in the chapter "Color Manager" in *Inside Macintosh: Advanced Color Imaging*) maintains an inverse table for every color table with which it works in order to speed up the process of searching the color table. If your application directly changes an entry in the color table, the Color Manager doesn't know that its inverse table no longer works correctly.

However, by using the routines `CTabChanged`, `PixPatChanged`, `PortChanged`, and `GDeviceChanged`, you can lessen the adverse effects of directly modifying the fields of `ColorTable`, `PixPat`, `GrafPort`, `CGrafPort`, and `GDevice` records. For example, should you directly change the field of a `ColorTable` record and then call the `CTabChanged` procedure, it invalidates the `ctSeed` field of the `ColorTable` record, which signals the Color Manager that the table has been changed and its inverse table needs to be rebuilt.

CTabChanged

If you modify the content of a `ColorTable` record (described on page 4-56), use the `CTabChanged` procedure.

```
PROCEDURE CTabChanged (ctab: CTabHandle);
```

`ctab` A handle to the `ColorTable` record changed by your application.

DESCRIPTION

For the `ColorTable` record you specify in the `ctab` parameter, the `CTabChanged` procedure calls the Color Manager function `GetCTSeed` to get a new seed (that is, a new, unique identifier in the `ctSeed` field of the `ColorTable` record) and notifies Color QuickDraw of the change.

SPECIAL CONSIDERATIONS

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

Your application should never need to directly modify a `ColorTable` record and use the `CTabChanged` procedure; instead, your application should use the QuickDraw routines described in this book for manipulating the values in a `ColorTable` record.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `CTabChanged` procedure are

Trap macro	Selector
`_QDExtensions`	$00040007

SEE ALSO

The `GetCTSeed` function is described in the chapter "Color Manager" in *Inside Macintosh: Advanced Color Imaging*.

PixPatChanged

If you modify the content of a `PixPat` record (described on page 4-58), including its `PixMap` record or the image in its `patData` field, use the `PixPatChanged` procedure.

```
PROCEDURE PixPatChanged (ppat: PixPatHandle);
```

`ppat` A handle to the changed pixel pattern.

DESCRIPTION

The `PixPatChanged` procedure sets the `patXValid` field of the `PixPat` record specified in the `ppat` parameter to –1 and notifies QuickDraw of the change.

If your application changes the `pmTable` *field* of a pixel pattern's `PixMap` record, it should call `PixPatChanged`. However, if your application changes the *content* of the color table referenced by the `PixMap` record's `pmTable` field, it should call `PixPatChanged` and the `CTabChanged` procedure as well. (The `CTabChanged` procedure is described in the preceding section.)

SPECIAL CONSIDERATIONS

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

Your application should never need to directly modify a `PixPat` record and use the `PixPatChanged` procedure; instead, your application should use the QuickDraw routines described in this book for manipulating the values in a `PixPat` record.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PixPatChanged` procedure are

Trap macro	Selector
`_QDExtensions`	$00040008

PortChanged

If you modify the content of a `GrafPort` record (described in the chapter "Basic QuickDraw" in this book) or `CGrafPort` record (described on page 4-48), including any of the data structures specified by handles within the record, use the `PortChanged` procedure.

```
PROCEDURE PortChanged (port: GrafPtr);
```

`port` A pointer to the `GrafPort` record that you have changed.

DESCRIPTION

The `PortChanged` procedure notifies QuickDraw that your application has changed the graphics port specified in the `port` parameter. If your application has changed a `CGrafPort` record, it must coerce its pointer (that is, its `CGrafPtr`) to a pointer to a `GrafPort` record (that is, to a `GrafPtr`) before passing the pointer in the `port` parameter.

You generally should not directly change any of the `PixPat` records specified in a `CGrafPort` record, but instead use the `PenPixPat` and `BackPixPat` procedures. However, if your application does change the content of a `PixPat` record, it should call the `PixPatChanged` procedure (described in the preceding section) as well as the `PortChanged` procedure.

If your application changes the `pmTable` field of the `PixMap` record specified in the graphics port, your application should call `PortChanged`. If your application changes the content of the `ColorTable` record referenced by the `pmTable` field, it should call `CTabChanged` as well.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PortChanged` procedure are

Trap macro	Selector
`_QDExtensions`	$00040009

GDeviceChanged

If you modify the content of a `GDevice` record (described in the chapter "Graphics Devices" in this book), use the `GDeviceChanged` procedure.

```
PROCEDURE GDeviceChanged (gdh: GDHandle);
```

DESCRIPTION

The `GDeviceChanged` procedure notifies Color QuickDraw that your application has changed the `GDevice` record specified in the `gdh` parameter.

If your application changes the `pmTable` *field* of the `PixMap` record specified in a `GDevice` record, your application should call `GDeviceChanged`. If your application changes the *content* of the `ColorTable` record referenced by the `PixMap` record, it should call `GDeviceChanged` and `CTabChanged` as well.

SPECIAL CONSIDERATIONS

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

Your application should never need to directly modify a `GDevice` record and use the `GDeviceChanged` procedure; instead, your application should use the QuickDraw routines described in this book for manipulating the values in a `GDevice` record.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GDeviceChanged` procedure are

Trap macro	Selector
`_QDExtensions`	$0004000A

Application-Defined Routine

You can customize the `SeedCFill` and `CalcCMask` procedures by writing your own color search function. For example, you might wish to use your own color search function to make `SeedCFill` generate a mask that allows filling around pixels that approximate the color of your seed point, rather than match it exactly.

The `SeedCFill` procedure generates a mask showing where the pixels in an image can be filled from a starting point, like the paint pouring from the MacPaint paint-bucket tool. The `CalcCMask` procedure generates a mask showing where pixels in an image *cannot* be filled from any of the outer edges of a rectangle you specify. You can then use these masks with the `CopyBits`, `CopyMask`, and `CopyDeepMask` procedures.

By default, `SeedCFill` returns 1's in the mask to indicate all pixels adjacent to a seed point whose colors do not exactly match the `RGBColor` record for the pixel at the seed point. By default, `CalcCMask` returns 1's in the mask to indicate what pixels have the exact RGB value that you specify in the `seedRGB` parameter, as well as which pixels are enclosed by shapes whose outlines consist entirely of pixels with this exact color. These procedures use a default color search function that matches exact colors.

You can customize these procedures by writing your own color search function and pointing to it in the `matchProc` parameters to these procedures, which then use your procedure instead of the default.

MyColorSearch

Here's how to declare a color search function to supply to the `SeedCFill` or `CalcCMask` procedure if you were to name the function `MyColorSearch`:

```
FUNCTION MyColorSearch (rgb: RGBColor;
                        position: LongInt): Boolean;
```

`rgb` The `RGBColor` record for a pixel.

`position` The position of the pixel within an image.

DESCRIPTION

Your color search function should analyze the `RGBColor` record passed to it in the `rgb` parameter. To mask a pixel approximating that color, your color search function should return `TRUE`; otherwise, it should return `FALSE`.

Your application should compare the `RGBColor` records that `SeedCFill` passes to your color search function against the `RGBColor` record for the pixel at the seed point you specify in that procedure's `seedH` and `seedV` parameters.

You can use a `MatchRec` record to determine the color of the seed pixel. When `SeedCFill` calls your color search function, the `GDRefCon` field of the current `GDevice` record (described in the chapter "Graphics Devices") contains a pointer to a `MatchRec` record that describes the seed point. This record has the following structure:

```
MatchRec =
   RECORD
      red:        Integer; {red component of seed pixel}
      green:      Integer; {green component of seed pixel}
      blue:       Integer; {blue component of seed pixel}
      matchData:  LongInt; {value in matchData parameter of }
                           { SeedCFill procedure}
   END;
```

The `matchData` field contains whatever value you pass to `SeedCFill` in the `matchData` parameter. In the `matchData` parameter, for instance, your application could pass the handle to a color table. Your color search function could then check whether the color for the pixel currently under analysis matches any of the colors in the table.

Similarly, your application should compare the colors that `CalcCMask` passes to your color search function against the color that you specify in that procedure's `seedRGB` parameter. When `CalcCMask` calls your color search function, the `GDRefCon` field of the current `GDevice` record (described in the chapter "Graphics Devices") contains a pointer to a `MatchRec` record for your seed color. The `matchData` field of this record contains whatever value you pass to `CalcCMask` in the `matchData` parameter.

Resources

This section describes the pixel pattern (`'ppat'`) resource, the color table (`'clut'`) resource, and the color icon (`'cicn'`) resource. Your application can use a `'ppat'` resource to create multicolored patterns for drawing. Your application can use a `'clut'` resource to define available colors for a pixel map or an indexed device. When you want to display a color icon within some element of your application (such as within a menu, an alert box, or a dialog box), you can create a `'cicn'` resource. These resource types should be marked as purgeable.

Note

These Color QuickDraw resources are compound structures and are more complex than a simple resource handle. When your application requests one of these resources, Color QuickDraw reads the requested resource, copies it, and then alters the copy before passing it to your application. Each time your application calls `GetPixPat`, for example, your application gets a new copy of a pixel pattern resource; therefore, your application should call `GetPixPat` only once for a particular pixel pattern resource. ◆

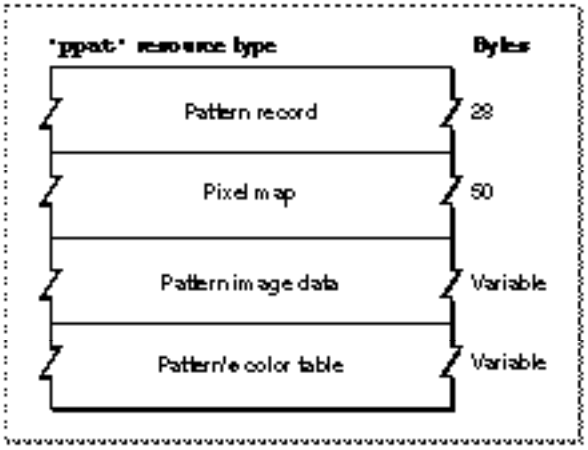
The Pixel Pattern Resource

You can use a pixel pattern resource to define a multicolored pattern to display with Color QuickDraw routines. A pixel pattern resource is a resource of type `'ppat'`. All `'ppat'` resources should be marked purgeable, and they must have resource IDs greater than 128. Use the `GetPixPat` function (described on page 4-88) to create a pixel pattern defined in a `'ppat'` resource. Color QuickDraw uses the information you specify to create a `PixPat` record in memory. (The `PixPat` record is described on page 4-58.)

This section describes the structure of this resource after it has been compiled by the Rez resource compiler, available from APDA. However, you typically use a high-level tool such as the ResEdit application, also available through APDA, to create `'ppat'` resources. You can then use the DeRez decompiler to convert your `'ppat'` resources into Rez input when necessary.

The compiled output format for a `'ppat'` resource is illustrated in Figure 4-16.

Figure 4-16 Format of a compiled pixel pattern (`'ppat'`) resource



The compiled version of a `'ppat'` resource contains the following elements:

- A pattern record. This is similar to the `PixPat` record (described on page 4-58), except that the resource contains an offset (rather than a handle) to a `PixMap` record (which is included in the resource), and it contains an offset (rather than a handle) to the pattern image data (which is also included in the resource).
- A pixel map. This is similar to the `PixMap` record (described on page 4-46), except that the resource contains an offset (rather than a handle) to a color table (which is included in the resource).
- Pattern image data. The size of the image data is calculated by subtracting the offset to the image data from the offset to the color table data.
- A color table. This follows the same format as the color table (`'clut'`) resource described next.

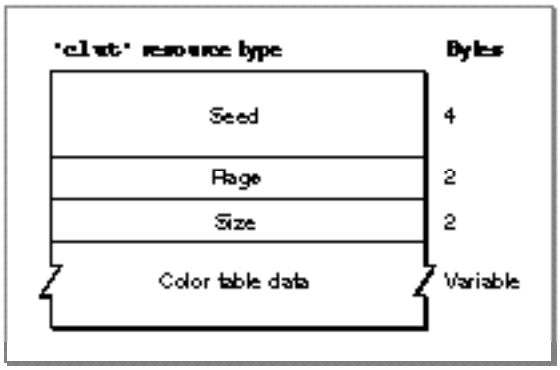
The Color Table Resource

You can use a color table resource to define a color table for a pixel pattern or an indexed device. To retrieve a color table stored in a color table resource, use the `GetCTable` function described on page 4-92. A color table resource is a resource of type `'clut'`. All `'clut'` resources should be marked purgeable, and they must have resource IDs greater than 128.

This section describes the structure of this resource after it has been compiled by the Rez resource compiler, available from APDA. However, you typically use a high-level tool such as the ResEdit application, also available through APDA, to create `'clut'` resources. You can then use the DeRez decompiler to convert your `'clut'` resources into Rez input when necessary.

The compiled output format for a `'clut'` resource is illustrated in Figure 4-17.

Figure 4-17 Format of a compiled color table (`'clut'`) resource



The compiled version of a `'clut'` resource contains the following elements:

- Seed. This contains the resource ID for this resource.
- Flags. A value of $0000 identifies this as a color table for a pixel map. A value of $8000 identifies this as a color table for an indexed device.
- Size. One less than the number of color specification entries in the rest of this resource.
- An array of color specification entries. Each entry contains a pixel value and a color specified by the values for the red, green, and blue components of the entry.

There are several default 'clut' resources for Macintosh computers containing 68020 and later processors. There is a default 'clut' resource for each of the standard pixel depths. The resource ID for each is the same as the pixel depth. For example, the default 'clut' resource for screens supporting 8 bits per pixel has a resource ID of 8.

Another default 'clut' resource defines the eight colors available for basic QuickDraw's eight-color system. This 'clut' resource has a resource ID of 127.

The Color Icon Resource

When you want to display a color icon within some element of your application (such as within a menu, an alert box, or a dialog box), you can create a color icon resource. A color icon resource is a resource of type 'cicn'. All color icon resources must be marked purgeable, and they must have resource IDs greater than 128. The 'cicn' resource was introduced with early versions of Color QuickDraw and is described here for completeness.

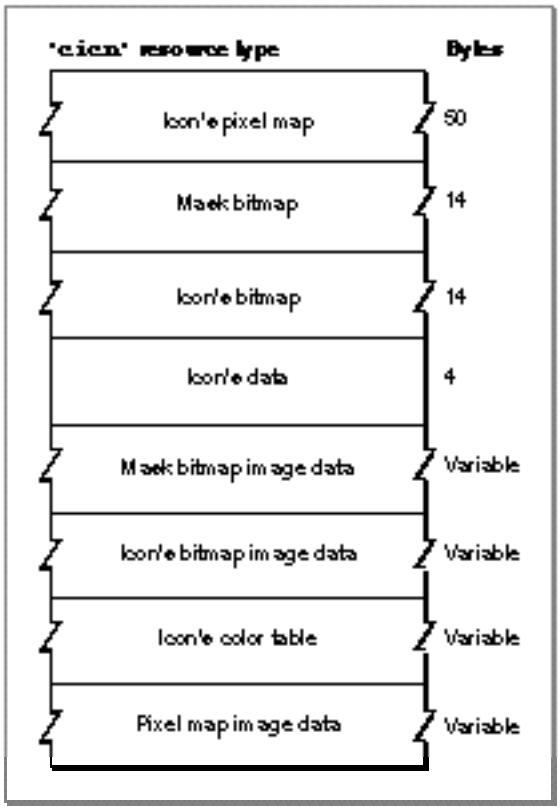
Using color icon resources, you can create icons similar to the ones that the Finder uses to display your application's files on the desktop; however, the Finder does *not* use or display any resources that you create of type 'cicn'. Instead, your application uses icon resources of type 'cicn' to display icons from within your application. (For information about the small and large 4-bit and 8-bit color icon resources—types 'ics4', 'icl4', 'ics8', and 'icl8'—necessary to define an icon family for the Finder's use, see *Inside Macintosh: Macintosh Toolbox Essentials*.)

Generally, you use color icon resources in menus, alert boxes, and dialog boxes, as described in the chapters "Menu Manager" and "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*. If you provide a color icon ('cicn') resource with the same resource ID as an icon ('ICON') resource, the Menu Manager and the Dialog Manager display the color icon instead of the black-and-white icon for users with color monitors. For information about drawing color icons without the aid of the Menu Manager or Dialog Manager (for example, to draw an icon in a window), see the chapter "Icon Utilities" in *Inside Macintosh: More Macintosh Toolbox*.

You can use a high-level tool such as the ResEdit application to create color icon resources. You can then use the DeRez decompiler to convert your color icon resources into Rez input when necessary.

The compiled output format for a `'cicn'` resource is illustrated in Figure 4-18.

Figure 4-18 Format of a compiled color icon (`'cicn'`) resource



The compiled version of a `'cicn'` resource contains the following elements:

- A pixel map. This pixel map describes the image when drawing the icon on a color screen.
- A bitmap for the icon's mask.
- A bitmap for the icon. This contains the image to use when drawing the icon to a 1-bit screen.
- Icon data.
- The bitmap image data for the icon's mask.
- The bitmap image data for the bitmap to be used on 1-bit screens. It may be `NIL`.
- A color table containing the color information for the icon's pixel map.
- The image data for the pixel map.

See the chapter "Icon Utilities" in *Inside Macintosh: More Macintosh Toolbox* for information about Macintosh Toolbox routines available to help you display icons.

Summary of Color QuickDraw

Pascal Summary

Constants

```
CONST
   {checking for Color QuickDraw and its features}
   gestaltQuickdrawVersion = 'qd  ';   {Gestalt selector for Color QuickDraw}
   gestalt8BitQD     = $100;  {8-bit Color QD}
   gestalt32BitQD    = $200;  {32-bit Color QD}
   gestalt32BitQD11  = $210;  {32-bit Color QDv1.1}
   gestalt32BitQD12  = $220;  {32-bit Color QDv1.2}
   gestalt32BitQD13  = $230;  {System 7: 32-bit Color QDv1.3}
   gestaltQuickdrawFeatures   = 'qdrw';   {Gestalt selector for Color }
                                          { QuickDraw features}
   gestaltHasColor         = 0;  {Color QuickDraw is present}
   gestaltHasDeepGWorlds   = 1;  {GWorlds deeper than 1 bit}
   gestaltHasDirectPixMaps = 2;  {PixMaps can be direct--16 or 32 bit}
   gestaltHasGrayishTextOr = 3;  {supports text mode grayishTextOr}
                  {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}

{special text transfer mode}
grayishTextOr = 49;

{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}
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}

   {transparent mode constant}
   transparent = 36; {replace the destination pixel with the source pixel }
                     { if the source pixel isn't equal to the background }
                     { color}

   hilite      = 50; {add to source or pattern mode for highlighting}
   hiliteBit   = 7;  {flag bit in HiliteMode (lowMem flag)}
   pHiliteBit  = 0;  {flag bit in HiliteMode used with BitClr procedure}

   defQDColors = 127;   {resource ID of 'clut' for default QDColors}

   {pixel type}
   RGBDirect = 16;      {16 & 32 bits per pixel pixelType value}

   {pmVersion values}
   baseAddr32 = 4;    {pixmap base address is 32-bit address}
```

Data Types

```
TYPE  PixMap      =
RECORD
   baseAddr:      Ptr;        {pixel image}
   rowBytes:      Integer;    {flags, and row width}
   bounds:        Rect;       {boundary rectangle}
   pmVersion:     Integer;    {PixMap record version number}
   packType:      Integer;    {packing format}
   packSize:      LongInt;    {size of data in packed state}
   hRes:          Fixed;      {horizontal resolution (dpi)}
   vRes:          Fixed;      {vertical resolution (dpi)}
   pixelType:     Integer;    {format of pixel image}
   pixelSize:     Integer;    {physical bits per pixel}
```

```
   cmpCount:      Integer;    {logical components per pixel}
   cmpSize:       Integer;    {logical bits per component}
   planeBytes:    LongInt;    {offset to next plane}
   pmTable:       CTabHandle; {handle to color table for this image}
   pmReserved:    LongInt;    {reserved for future expansion}
END;

CGrafPtr       = ^CGrafPort;
CGrafPort      =
RECORD
   device:        Integer;       {device ID for font selection}
   portPixMap:    PixMapHandle;  {handle to PixMap record}
   portVersion:   Integer;       {highest 2 bits always set}
   grafVars:      Handle;        {handle to GrafVars record}
   chExtra:       Integer;       {added width for nonspace characters}
   pnLocHFrac:    Integer;       {pen fraction}
   portRect:      Rect;          {port rectangle}
   visRgn:        RgnHandle;     {visible region}
   clipRgn:       RgnHandle;     {clipping region}
   bkPixPat:      PixPatHandle;  {background pattern}
   rgbFgColor:    RGBColor;      {requested foreground color}
   rgbBkColor:    RGBColor;      {requested background color}
   pnLoc:         Point;         {pen location}
   pnSize:        Point;         {pen size}
   pnMode:        Integer;       {pattern mode}
   pnPixPat:      PixPatHandle;  {pen pattern}
   fillPixPat:    PixPatHandle;  {fill pattern}
   pnVis:         Integer;       {pen visibility}
   txFont:        Integer;       {font number for text}
   txFace:        Style;         {text's font style}
   txMode:        Integer;       {source mode for text}
   txSize:        Integer;       {font size for text}
   spExtra:       Fixed;         {added width for space characters}
   fgColor:       LongInt;       {actual foreground color}
   bkColor:       LongInt;       {actual background color}
   colrBit:       Integer;       {plane being drawn}
   patStretch:    Integer;       {used internally}
   picSave:       Handle;        {picture being saved, used internally}
   rgnSave:       Handle;        {region being saved, used internally}
   polySave:      Handle;        {polygon being saved, used internally}
   grafProcs:     CQDProcsPtr;   {low-level drawing routines}
END;
```

```
RGBColor       =
RECORD
   red:     Integer;    {red component}
   green:   Integer;    {green component}
   blue:    Integer;    {blue component}
END;

ColorSpec      =
RECORD
   value:   Integer;    {index or other value}
   rgb:     RGBColor;   {true color}
END;

cSpecArray : ARRAY[0..0] OF ColorSpec;

CTabHandle     = ^CTabPtr;
CTabPtr        = ^ColorTable;
ColorTable     =
RECORD
   ctSeed:  LongInt;    {unique identifier from table}
   ctFlags: Integer;    {contains flags describing the ctTable field; }
                        { clear for a PixMap record}
   ctSize:  Integer;    {number of entries in the next field minus 1}
   ctTable: cSpecArray; {an array of ColorSpec records}
END;

MatchRec       =
RECORD
   red:        Integer; {red component of seed}
   green:      Integer; {green component of seed}
   blue:       Integer; {blue component of seed}
   matchData:  LongInt; {value in matchData parameter of }
                        { SeedCFill or CalcCMask}
END;
```

```
PixPatHandle   = ^PixPatPtr;
PixPatPtr      = ^PixPat;
PixPat         =
RECORD
   patType:    Integer;       {pattern type}
   patMap:     PixMapHandle;  {PixMap record for pattern}
   patData:    Handle;        {pixel image defining pattern}
   patXData:   Handle;        {expanded pixel image}
   patXValid:  Integer;       {flags for expanded pattern data}
   patXMap:    Handle;        {handle to expanded pattern data}
   pat1Data:   Pattern;       {bit pattern for a GrafPort record}
END;

CQDProcsPtr       = ^CQDProcs
CQDProcs          =
RECORD
   textProc:      Ptr;  {text drawing}
   lineProc:      Ptr;  {line drawing}
   rectProc:      Ptr;  {rectangle drawing}
   rRectProc:     Ptr;  {rounded rectangle drawing}
   ovalProc:      Ptr;  {oval drawing}
   arcProc:       Ptr;  {arc and 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}
   opcodeProc:    Ptr;  {reserved for future use}
   newProc1:      Ptr;  {reserved for future use}
   newProc2:      Ptr;  {reserved for future use}
   newProc3:      Ptr;  {reserved for future use}
   newProc4:      Ptr;  {reserved for future use}
   newProc5:      Ptr;  {reserved for future use}
   newProc6:      Ptr;  {reserved for future use}
END;
```

```
GrafVars      =
RECORD
   rgbOpColor:       RGBColor;   {color for addPin, subPin, and blend}
   rgbHiliteColor:   RGBColor;   {color for highlighting}
   pmFgColor:        Handle;     {palette handle for foreground color}
   pmFgIndex:        Integer;    {index value for foreground}
   pmBkColor:        Handle;     {palette handle for background color}
   pmBkIndex:        Integer;    {index value for background}
   pmFlags:          Integer;    {flags for Palette Manager}
END;
```

Color QuickDraw Routines

Opening and Closing Color Graphics Ports

```
PROCEDURE OpenCPort         (port: CGrafPtr);
PROCEDURE InitCPort         (port: CGrafPtr);
PROCEDURE CloseCPort        (port: CGrafPtr);
```

Managing a Color Graphics Pen

```
PROCEDURE PenPixPat         (ppat: PixPatHandle);
```

Changing the Background Pixel Pattern

```
PROCEDURE BackPixPat        (ppat: PixPatHandle);
```

Drawing With Color QuickDraw Colors

```
PROCEDURE RGBForeColor      (color: RGBColor);
PROCEDURE RGBBackColor      (color: RGBColor);
PROCEDURE SetCPixel         (h,v: Integer; cPix: RGBColor);
PROCEDURE FillCRect         (r: Rect; ppat: PixPatHandle);
PROCEDURE FillCRoundRect    (r: Rect; ovalWidth,ovalHeight: Integer;
                             ppat: PixPatHandle);
PROCEDURE FillCOval         (r: Rect; ppat: PixPatHandle);
PROCEDURE FillCArc          (r: Rect; startAngle,arcAngle: Integer;
                             ppat: PixPatHandle);
```

```
PROCEDURE FillCPoly         (poly: PolyHandle; ppat: PixPatHandle);
PROCEDURE FillCRgn          (rgn: RgnHandle; ppat: PixPatHandle);
PROCEDURE OpColor           (color: RGBColor);
PROCEDURE HiliteColor       (color: RGBColor);
```

Determining Current Colors and Best Intermediate Colors

```
PROCEDURE GetForeColor      (VAR color: RGBColor);
PROCEDURE GetBackColor      (VAR color: RGBColor);
PROCEDURE GetCPixel         (h,v: Integer; VAR cPix: RGBColor);
FUNCTION GetGray            (device: GDHandle; backGround: RGBColor;
                             VAR foreGround: RGBColor): Boolean;
```

Calculating Color Fills

```
PROCEDURE SeedCFill         (srcBits,dstBits: BitMap;
                             srcRect,dstRect: Rect; seedH,seedV: Integer;
                             matchProc: ProcPtr; matchData: LongInt);
PROCEDURE CalcCMask         (srcBits,dstBits: BitMap;
                             srcRect,dstRect: Rect; seedRGB: RGBColor;
                             matchProc: ProcPtr; matchData: LongInt);
```

Creating, Setting, and Disposing of Pixel Maps

```
{DisposePixMap is also spelled as DisposPixMap}
FUNCTION NewPixMap          : PixMapHandle;
PROCEDURE CopyPixMap        (srcPM,dstPM: PixMapHandle);
PROCEDURE SetPortPix        (pm: PixMapHandle);
PROCEDURE DisposePixMap     (pm: PixMapHandle);
```

Creating and Disposing of Pixel Patterns

```
{DisposePixPat is also spelled as DisposPixPat}
FUNCTION GetPixPat          (patID: Integer): PixPatHandle;
FUNCTION NewPixPat          : PixPatHandle;
PROCEDURE CopyPixPat        (srcPP,dstPP: PixPatHandle);
PROCEDURE MakeRGBPat        (ppat: PixPatHandle; myColor: RGBColor);
PROCEDURE DisposePixPat     (ppat: PixPatHandle);
```

Creating and Disposing of Color Tables

```
{DisposeCTable is also spelled as DisposCTable}
FUNCTION GetCTable          (ctID: Integer): CTabHandle;
PROCEDURE DisposeCTable     (cTable: CTabHandle);
```

Retrieving Color QuickDraw Result Codes

```
FUNCTION QDError:           Integer;
```

Customizing Color QuickDraw Operations

```
PROCEDURE SetStdCProcs      (VAR cProcs: CQDProcs);
```

Reporting Data Structure Changes to QuickDraw

```
PROCEDURE CTabChanged       (ctab: CTabHandle);
PROCEDURE PixPatChanged     (ppat: PixPatHandle);
PROCEDURE PortChanged       (port: GrafPtr);
PROCEDURE GDeviceChanged    (gdh: GDHandle);
```

Application-Defined Routine

```
FUNCTION MyColorSearch      (rgb: RGBColor; position: LongInt): Boolean;
```

C Summary

Constants

```
enum {
   /* checking for Color QuickDraw and its features */
   gestaltQuickdrawVersion = 'qd  ',   /* Gestalt selector for Color
                                          QuickDraw */
   gestalt8BitQD     = 0x100, /* 8-bit Color QD */
   gestalt32BitQD    = 0x200, /* 32-bit Color QD */
   gestalt32BitQD11  = 0x210, /* 32-bit Color QDv1.1 */
   gestalt32BitQD12  = 0x220, /* 32-bit Color QDv1.2 */
   gestalt32BitQD13  = 0x230, /* System 7: 32-bit Color QDv1.3 */
   gestaltQuickdrawFeatures
                     = 'qdrw',   /* Gestalt selector for Color QuickDraw
                                    features */
   gestaltHasColor         = 0,  /* Color QuickDraw is present */
```

```
gestaltHasDeepGWorlds   = 1,  /* GWorlds deeper than 1 bit */
gestaltHasDirectPixMaps = 2,  /* PixMaps can be direct--16 or 32 bit */
gestaltHasGrayishTextOr = 3,  /* supports text mode grayishTextOr */

/* 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 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 */

/* special text transfer mode */
grayishTextOr  = 49,
```

```
/* 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,
                     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 */
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 */

/* transparent mode constant */
transparent = 36, /* replace the destination pixel with the source pixel
                     if the source pixel isn't equal to the background
                     color */
```

```
   hilite      = 50, /* add to source or pattern mode for highlighting */
   hiliteBit   = 7,  /* flag bit in highlight mode (lowMem flag) */
   pHiliteBit  = 0,  /* flag bit in highlight mode used with BitClr
                        procedure */

   defQDColors    = 127,   /* resource ID of 'clut' for default QDColors */

   /* pixel type */
   RGBDirect = 16,          /* 16 & 32 bits/pixel pixelType value */

   /* pmVersion values */
   baseAddr32 = 4,         /* pixel map base address is 32-bit address */

};
```

Data Types

```
struct PixMap {
   Ptr         baseAddr;      /* pixel image */
   short       rowBytes;      /* flags, and row width */
   Rect        bounds;        /* boundary rectangle */
   short       pmVersion;     /* PixMap version number */
   short       packType;      /* packing format */
   long        packSize;      /* size of data in packed state */
   Fixed       hRes;          /* horizontal resolution (dpi) */
   Fixed       vRes;          /* vertical resolution (dpi) */
   short       pixelType;     /* format of pixel image */
   short       pixelSize;     /* physical bits per pixel */
   short       cmpCount;      /* logical components per pixel */
   short       cmpSize;       /* logical bits per component */
   long        planeBytes;    /* offset to next plane */
   CTabHandle  pmTable;       /* handle to the ColorTable struct */
   long        pmReserved;    /* reserved for future expansion; must be 0 */
};
typedef struct PixMap PixMap;
typedef PixMap *PixMapPtr, **PixMapHandle;

typedef unsigned char PixelType;
```

```
struct CGrafPort {
   short          device;     /* device ID for font selection */
   PixMapHandle   portPixMap; /* handle to PixMap struct */
   short          portVersion;/* highest 2 bits always set */
   Handle         grafVars;   /* handle to a GrafVars struct */
   short          chExtra;    /* added width for nonspace characters */
   short          pnLocHFrac; /* pen fraction */
   Rect           portRect;   /* port rectangle */
   RgnHandle      visRgn;     /* visible region */
   RgnHandle      clipRgn;    /* clipping region */
   PixPatHandle   bkPixPat;   /* background pattern */
   RGBColor       rgbFgColor; /* requested foreground color */
   RGBColor       rgbBkColor; /* requested background color */
   Point          pnLoc;      /* pen location */
   Point          pnSize;     /* pen size */
   short          pnMode;     /* pattern mode */
   PixPatHandle   pnPixPat;   /* pen pattern */
   PixPatHandle   fillPixPat; /* fill pattern */
   short          pnVis;      /* pen visibility */
   short          txFont;     /* font number for text */
   Style          txFace;     /* text's font style */
   char           filler;
   short          txMode;     /* source mode for text */
   short          txSize;     /* font size for text */
   Fixed          spExtra;    /* added width for space characters */
   long           fgColor;    /* actual foreground color */
   long           bkColor;    /* actual background color */
   short          colrBit;    /* plane being drawn */
   short          patStretch; /* used internally */
   Handle         picSave;    /* picture being saved, used internally */
   Handle         rgnSave;    /* region being saved, used internally */
   Handle         polySave;   /* polygon being saved, used internally */
   CQDProcsPtr    grafProcs;  /* low-level drawing routines */
};
typedef struct CGrafPort CGrafPort;
typedef CGrafPort *CGrafPtr;
typedef CGrafPtr CWindowPtr;
```

```
struct RGBColor {
   unsigned short red;     /* magnitude of red component */
   unsigned short green;   /* magnitude of green component */
   unsigned short blue;    /* magnitude of blue component */
};
typedef struct RGBColor RGBColor;

struct ColorSpec {
   short       value;   /* index or other value */
   RGBColor    rgb;     /* true color */
};
typedef struct ColorSpec ColorSpec;
typedef ColorSpec *ColorSpecPtr;
typedef ColorSpec CSpecArray[1];

struct ColorTable {
   long        ctSeed;     /* unique identifier for table */
   short       ctFlags;    /* high bit: 0 = PixMap; 1 = device */
   short       ctSize;     /* number of entries in next field */
   CSpecArray  ctTable;    /* array[0..0] of ColorSpec records */
};
typedef struct ColorTable ColorTable;
typedef ColorTable *CTabPtr, **CTabHandle;

struct MatchRec {
   unsigned short red;        /* red component of seed */
   unsigned short green;      /* green component of seed */
   unsigned short blue;       /* blue component of seed */
   long matchData;            /* value in matchData parameter of
                                 SeedCFill or CalcCMask */
};

typedef struct MatchRec MatchRec;

struct PixPat {
   short          patType;    /* pattern type */
   PixMapHandle   patMap;     /* PixMap structure for pattern */
   Handle         patData;    /* pixel-image defining pattern */
   Handle         patXData;   /* expanded pattern image */
   short          patXValid;  /* for expanded pattern data */
   Handle         patXMap;    /* handle to expanded pattern data */
   Pattern        pat1Data;   /* a bit pattern for a GrafPort structure */
};
```

```
typedef struct PixPat PixPat;
typedef PixPat *PixPatPtr, **PixPatHandle;

struct CQDProcs {
   Ptr textProc;     /* text drawing */
   Ptr lineProc;     /* line drawing */
   Ptr rectProc;     /* rectangle drawing */
   Ptr rRectProc;    /* rounded rectangle drawing */
   Ptr ovalProc;     /* oval drawing */
   Ptr arcProc;      /* arc/wedge drawing */
   Ptr polyProc;     /* polygon 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 */
   Ptr opcodeProc;   /* reserved for future use */
   Ptr newProc1;     /* reserved for future use */
   Ptr newProc2;     /* reserved for future use */
   Ptr newProc3;     /* reserved for future use */
   Ptr newProc4;     /* reserved for future use */
   Ptr newProc5;     /* reserved for future use */
   Ptr newProc6;     /* reserved for future use */
};
typedef struct CQDProcs CQDProcs;
typedef CQDProcs *CQDProcsPtr;

struct GrafVars {
   RGBColor    rgbOpColor;       /* color for addPin,subPin,and blend */
   RGBColor    rgbHiliteColor;   /* color for highlighting */
   Handle      pmFgColor;        /* palette handle for foreground color */
   short       pmFgIndex;        /* index value for foreground */
   Handle      pmBkColor;        /* palette handle for background color */
   short       pmBkIndex;        /* index value for background */
   short       pmFlags;          /* flags for Palette Manager */
};
typedef struct GrafVars GrafVars;
typedef GrafVars *GVarPtr, **GVarHandle;
```

Color QuickDraw Functions

Opening and Closing Color Graphics Ports

```
pascal void OpenCPort       (CGrafPtr port);
pascal void InitCPort       (CGrafPtr port);
pascal void CloseCPort      (CGrafPtr port);
```

Managing a Color Graphics Pen

```
pascal void PenPixPat       (PixPatHandle pp);
```

Changing the Background Pixel Pattern

```
pascal void BackPixPat      (PixPatHandle pp);
```

Drawing With Color QuickDraw Colors

```
pascal void RGBForeColor    (const RGBColor *color);
pascal void RGBBackColor    (const RGBColor *color);
pascal void SetCPixel       (short h, short v, const RGBColor *cPix);
pascal void FillCRect       (const Rect *r, PixPatHandle pp);
pascal void FillCRoundRect  (const Rect *r, short ovalWidth,
                             short ovalHeight, PixPatHandle pp);
pascal void FillCOval       (const Rect *r, PixPatHandle pp);
pascal void FillCArc        (const Rect *r, short startAngle,
                             short arcAngle, PixPatHandle pp);
pascal void FillCPoly       (PolyHandle poly, PixPatHandle pp);
pascal void FillCRgn        (RgnHandle rgn, PixPatHandle pp);
pascal void OpColor         (const RGBColor *color);
pascal void HiliteColor     (const RGBColor *color);
```

Determining Current Colors and Best Intermediate Colors

```
pascal void GetForeColor    (RGBColor *color);
pascal void GetBackColor    (RGBColor *color);
pascal void GetCPixel       (short h, short v, RGBColor *cPix);
pascal Boolean GetGray      (GDHandle device, const RGBColor *backGround,
                             RGBColor *foreGround);
```

Calculating Color Fills

```
pascal void SeedCFill       (const BitMap *srcBits, const BitMap *dstBits,
                             const Rect *srcRect, const Rect *dstRect,
                             short seedH, short seedV,
                             ColorSearchProcPtr matchProc, long matchData);
pascal void CalcCMask       (const BitMap *srcBits, const BitMap *dstBits,
                             const Rect *srcRect, const Rect *dstRect,
                             const RGBColor *seedRGB,
                             ColorSearchProcPtr matchProc, long matchData);
```

Creating, Setting, and Disposing of Pixel Maps

```
/* DisposePixMap is also spelled as DisposPixMap */
pascal PixMapHandle NewPixMap
                            (void);
pascal void CopyPixMap      (PixMapHandle srcPM, PixMapHandle dstPM);
pascal void SetPortPix      (PixMapHandle pm);
pascal void DisposePixMap   (PixMapHandle pm);
```

Creating and Disposing of Pixel Patterns

```
/* DisposePixPat is also spelled as DisposPixPat */
pascal PixPatHandle GetPixPat
                            (short patID);
pascal PixPatHandle NewPixPat
                            (void);
pascal void CopyPixPat      (PixPatHandle srcPP, PixPatHandle dstPP);
pascal void MakeRGBPat      (PixPatHandle pp, const RGBColor *myColor);
pascal void DisposePixPat   (PixPatHandle pp);
```

Creating and Disposing of Color Tables

```
/* DisposeCTable is also spelled as DisposCTable */
pascal CTabHandle GetCTable
                            (short ctID);
pascal void DisposeCTable   (CTabHandle cTable);
```

Retrieving Color QuickDraw Result Codes

```
pascal short QDError        (void);
```

Customizing Color QuickDraw Operations

```
pascal void SetStdCProcs    (CQDProcs *procs);
```

Reporting Data Structure Changes to QuickDraw

```
pascal void CTabChanged     (CTabHandle ctab);
pascal void PixPatChanged   (PixPatHandle ppat);
pascal void PortChanged     (GrafPtr port);
pascal void GDeviceChanged  (GDHandle gdh);
```

Application-Defined Function

```
pascal Boolean MyColorSearch(rgb RGBColor, position LongInt);
```

Assembly-Language Summary

Data Structures

PixMap Data Structure

0	`pmBaseAddr`	long	pixel image
4	`pmRowBytes`	word	flags, and row width
6	`pmBounds`	8 bytes	boundary rectangle
14	`pmVersion`	word	`PixMap` version number
16	`pmPackType`	word	packing format
18	`pmPackSize`	long	size of data in packed state
22	`pmHRes`	long	horizontal resolution (dpi)
26	`pmVRes`	long	vertical resolution (dpi)
30	`pmPixelType`	word	format of pixel image
32	`pmPixelSize`	word	physical bits per pixel
34	`pmCmpCount`	word	logical components per pixel
36	`pmCmpSize`	word	logical bits per component
38	`pmPlaneBytes`	long	offset to next plane
42	`pmTable`	long	handle to next `ColorTable` record
46	`pmReserved`	long	reserved; must be 0

CGrafPort Data Structure

0	device	short	device ID for font selection
2	portPixMap	long	handle to PixMap record
6	portVersion	short	highest 2 bits always set
8	grafVars	long	handle to GrafVars record
12	chExtra	short	added width for nonspace characters
14	pnLocHFrac	short	pen fraction
16	portRect	8 bytes	port rectangle
24	visRgn	long	visible region
28	clipRgn	long	clipping region
32	bkPixPat	long	background pattern
36	rgbForeColor	6 bytes	requested foreground color
42	rgbBackColor	6 bytes	requested background color
48	pnLoc	long	pen location
52	pnSize	long	pen size
56	pnMode	word	pattern mode
58	pnPixPat	long	pen pattern
62	fillPixPat	long	fill pattern
66	pnVis	word	pen visibility
68	txFont	word	font number for text
70	txFace	word	text's font style
72	txMode	word	source mode for text
74	txSize	word	font size for text
76	spExtra	long	added width for space characters
80	fgColor	long	actual foreground color
84	bkColor	long	actual background color
88	colrBit	word	plane being drawn
90	patStretch	word	used internally
92	picSave	long	picture being saved, used internally
96	rgnSave	long	region being saved, used internally
100	polySave	long	polygon being saved, used internally
104	grafProcs	long	low-level drawing routines

Relative Offsets of Additional Fields in a CGrafPort Record

portBits	portPixMap	long	handle to PixMap record
portPixMap+4	portVersion	word	highest 2 bits always set
portVersion+2	grafVars	long	handle to a GrafVars record
grafVars+4	chExtra	word	added width for nonspace characters
chExtra+2	pnLocHFrac	word	pen fraction
bkPat	bkPixPat	long	background pattern
bkPixPat+4	rgbFgColor	6 bytes	requested foreground color
rgbFgColor+6	rgbBkColor	6 bytes	requested background color
pnPat	pnPixPat	long	pen pattern
pnPixPat+4	fillPixPat	long	fill pattern

RGBColor Data Structure

0	`red`	short	magnitude of red component
2	`green`	short	magnitude of green component
4	`blue`	short	magnitude of blue component

ColorSpec Data Structure

0	`value`	short	index or other value
2	`rgb`	6 bytes	true color

ColorTable Data Structure

0	`ctSeed`	long	unique identifier for table
4	`transIndex`	word	index of transparent pixel (obsolete)
8	`ctFlags`	word	high bit: 0 = pixel map; 1 = device
10	`ctSize`	word	number of entries in next field
12	`ctTable`	variable	array of `ColorSpec` records

MatchRec Data Structure

0	`red`	word	red component of seed
2	`green`	word	green component of seed
4	`blue`	word	blue component of seed
6	`matchData`	long	value in `matchData` parameter of `SeedCFill` or `CalcCMask`

PixPat Data Structure

0	`patType`	word	pattern type
2	`patMap`	long	handle to `PixMap` record for pattern
6	`patData`	long	pixel-image defining pattern
10	`patXData`	long	expanded pattern data
14	`patXValid`	word	for expanded pattern data
16	`patXMap`	long	handle to expanded pattern data
20	`pat1Data`	8 bytes	a bit pattern for a `GrafPort` record

CQDProcs 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
52	opcodeProc	long	reserved for future use
56	newProc1	long	reserved for future use
60	newProc2	long	reserved for future use
64	newProc3	long	reserved for future use
68	newProc4	long	reserved for future use
72	newProc5	long	reserved for future use
76	newProc6	long	reserved for future use

GrafVars Data Structure

0	rgbOpColor	6 bytes	color for addPin, subPin, and blend
6	rgbHiliteColor	6 bytes	color for highlighting
12	pmFgColor	long	palette handle for foreground color
16	pmFgIndex	short	index value for foreground color
18	pmBkColor	long	palette handle for background color
22	pmBkIndex	short	index value for background color
24	pmFlags	short	flags for Palette Manager

Trap Macros Requiring Routine Selectors

_QDExtensions

Selector	Routine
$00040007	CTabChanged
$00040008	PixPatChanged
$00040009	PortChanged
$0004000A	GDeviceChanged

Result Codes

noErr	0	No error
paramErr	–50	Illegal parameter to NewGWorld
	–143	CopyBits couldn't allocate required temporary memory
	–144	Ran out of stack space while drawing polygon
noMemForPictPlaybackErr	–145	Insufficient memory for drawing the picture
regionTooBigError	–147	Region too big or complex
pixmapTooDeepErr	–148	Pixel map is deeper than 1 bit per pixel
nsStackErr	–149	Insufficient stack
cMatchErr	–150	Color2Index failed to find an index
cTempMemErr	–151	Failed to allocate memory for temporary structures
cNoMemErr	–152	Failed to allocate memory for structure
cRangeErr	–153	Range error on color table request
cProtectErr	–154	ColorTable record entry protection violation
cDevErr	–155	Invalid type of graphics device
cResErr	–156	Invalid resolution for MakeITable
cDepthErr	–157	Invalid pixel depth specified to NewGWorld
rgnTooBigErr	–500	Bitmap would convert to a region greater than 64 KB