11

QUICKDRAW PRELIMINARIES

Demonstration Program: PreQuickDraw

QuickDraw and Imaging

QuickDraw is a collection of system software routines that your application uses to perform **imaging** operations, that is, the construction and display of graphical information for display on output devices such as screens and printers.

This chapter serves as a prelude to Chapter 12, and introduces certain matters which need to be discussed before the matter of actually drawing with QuickDraw is addressed. These matters include RGB colours, colour and the video device, the graphics port, translation of RGB values, and graphics devices.

RGB Colours and Pixels

In QuickDraw, colours are specified as **RGB colours** using an `RGBColor` structure:

```
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;
```

Note that an RGB colour is defined by three components (red, green and blue). When the `red`, `green` and `blue` fields of the `RGBColor` structure are assigned the maximum possible value (`0xFFFF`), the resulting colour is white. When these fields are assigned the minimum value (`0x0000`), the resulting colour is black.

A **pixel** (picture element) is the smallest dot that QuickDraw can draw. Each colour pixel represents up to 48 bits in memory.

Colour and the Video Device

QuickDraw supports a variety of screens of differing sizes and colour capabilities, and is thus device-independent. Accordingly, you do not have to concern yourself with the capabilities of individual screens. For example, when your application uses an `RGBColor` structure to specify a colour by its red, green and blue components, with each component defined in a 16-bit integer, QuickDraw compares the resulting 48-bit value with the colours actually available on a video device (such as a plug-in video card or a built-in video interface) at execution time and then chooses the closest match. What the user finally sees depends on the characteristics of the actual video device and screen.

The video device that controls a screen may have either:

- **Indexed colours**, which support pixels of 1-bit, 2-bit, 4-bit, or 8-bit pixel depths[1]. The indexed colour system was introduced with the Macintosh II, that is, at a time when memory was scarce and moving megabyte images around was quite impractical.
- **Direct colours**, which support pixels of 16-bit and 32-bit depths. Most video devices in the current day are direct colour devices. (However, as will be seen, there are circumstances in which a direct colour device will act like an indexed colour device.)

QuickDraw automatically determines which method is used by the video device and matches your requested 48-bit colour with the closest available colour.

Indexed Colour Devices

Video devices using indexed colours support a maximum of 256 colours at any one time, that is, with indexed colour, the maximum value of a pixel is limited to a single byte, with each pixel's byte specifying one of 256 different values.

Video devices implementing indexed colour contain a data structure called a **colour lookup table** (**CLUT**), which contains entries for all possible colour values. Most indexed video devices use a **variable CLUT**, which allows your application to load the CLUT with different sets of colours depending on the image being displayed.

When your application uses a 48-bit `RGBColor` structure to specify a colour, the Color Manager compares the CLUT entries on the video device with the specified `RGBColor` colour, determines which colour in the CLUT is closest, and passes QuickDraw the index to this colour. This is the colour that QuickDraw draws with. Fig 1 illustrates this process.

1. The application calls a QuickDraw routine to draw an object in a colour specified in an RGBColor structure.

2. QuickDraw calls the Color Manager to ascertain which colour in the device's CLUT is closest to the requested colour.

3. The index value representing the best match is returned to Color QuickDraw.

4. QuickDraw puts the index value in the appropriate places in video RAM.

5. The video device continually displays video RAM by taking the index values, converting them to colours according to the CLUT entries at those indexes, and sending them to the screen via digital-to-analog converters.

Note: The system creates and initialises a GDevice structure for each video device found during startup. The colour table within this structure is kept synchronised with the video card's CLUT.

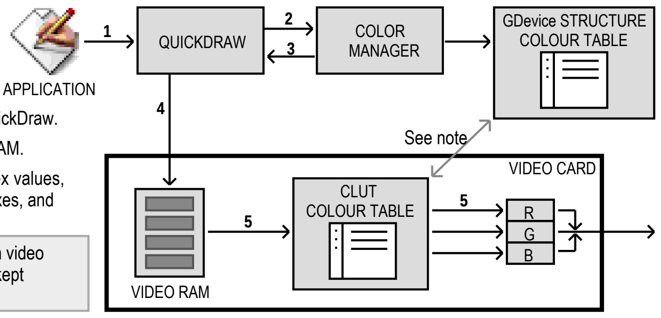


FIG 1 - INDEXED COLOUR SYSTEM

Direct Colour Devices

Video devices which implement direct colour eliminate the competition for limited colour lookup table spaces and remove the need for colour table matching. By using direct colour, video devices can support thousands or millions of colours.

When you specify a colour using a 48-bit `RGBColor` structure on a direct colour system, QuickDraw truncates the least significant bits of its red, green and blue components to either 16 bits (five bits each for red, green and blue, with one bit unused) or 32 bits (eight bits for red, green and blue, with eight bits unused). (See Translation of RGB Colours to Pixel Values, below.) Using 16 bits, direct video devices can display 32,768 different colours. Using 32 bits, the device can display 16,777,215 different colours

Fig 2 illustrates the direct colour system.

[1] Pixel depth means the number of bits assigned to each pixel, and thus determines the maximum number of colours that can be displayed at the one time. A 4-bit pixel depth, for example, means that an individual pixel can be displayed in any one of 16 separate colours. An 8-bit pixel depth means that an individual pixel can be displayed in any one of 256 separate colours.

1. The application calls a QuickDraw routine to draw an object in a colour specified in an RGBColor structure.

2. QuickDraw knows from the GDevice structure that the screen is controlled by a direct device in which pixels are, say, 32 bits deep, which meens that eight bits are used for each of the red, green, and blue components of the requested colour.

3. Accordingly, QuickDraw passes the high eight bits from each 16-bit component of the 48-bit RGBColor structure to the video device, which stores the resulting 24-bit value in video RAM for the object.

4. The video device continually displayes video RAM by sending the 8-bit red, green, and blue values for the colour to digital-to-analog converters which produce a signal for the screen

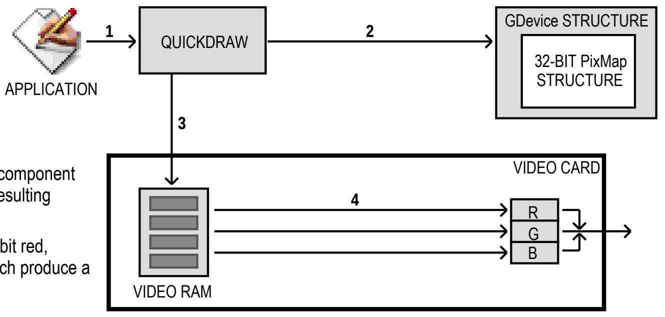


FIG 2 - DIRECT COLOUR SYSTEM

Direct colour not only removes much of the complexity of the CLUT mechanism for video device developers, but also allows the display of thousands or millions of colours simultaneously, resulting in near-photographic resolution.

Direct Devices Operating Like Indexed Devices

Note that, when a user sets a direct colour device to use 256 colours (or less) as either a grayscale or colour device, the direct device creates a CLUT and operates like an indexed device.

Graphics Port

A graphics port[2] defines a complete drawing environment. Amongst other things, a graphics port:

- Contains a handle to a **pixel map** which, in turn, contains a pointer to the area of memory in which your drawing operations take place.
- Contains a metaphorical graphics **pen** with which to perform drawing operations. (You can set this pen to different sizes, patterns and colours.)
- Holds information about text, which is styled and sized according to information in the graphics port.

The information in a graphics port is maintained by QuickDraw.

The graphics port is an opaque data structure. The data types `CGrafPtr` and `GrafPtr` are defined as pointers to such objects:

```
typedef struct OpaqueGrafPtr* GrafPtr;
typedef GrafPtr CGrafPtr;
```

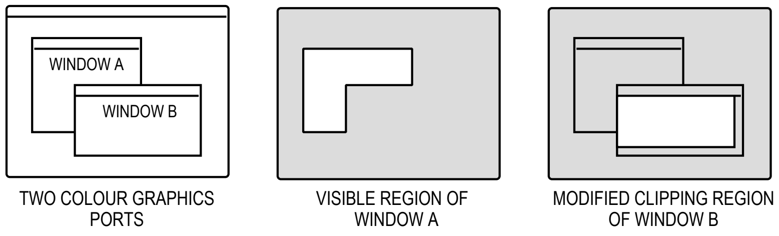
Accessor Functions

Accessor functions are provided to access the information in colour graphic port objects. The main accessor functions are as follows:

Accessor Function	*Description*
`GetPortPixMap`	Get a handle to the graphics port's pixel map
`GetPortBounds` `SetPortBounds`	Get and set the graphics port rectangle. Your application's drawing operations take place inside the port rectangle (which, for a window's graphics port is also called the **content region**.) The port rectangle uses a local coordinate system in which the upper-left corner of the port rectangle has a vertical coordinate of 0 and a horizontal coordinate of 0.
`GetPortVisRegion` `SetPortVisRegion`	Get and set the visible region. The **visible region** (which, by default, is equivalent to the port rectangle) is the region of the

[2] The term "graphics port" originally pertained to the one-bit graphics port used by the early black-and-white Macintoshes. The colour graphics port was introduced when colour came to the Macintosh with the Macintosh II. In the Carbon era, the black-and-white graphics port is irrelevant. Accordingly, where the term "graphics port" is used in this book, a colour graphics port should be assumed unless otherwise stated.

	graphics port that is actually visible on screen (see Fig 3).

`GetPortClipRegion`
`SetPortClipRegion`
`SetClip`
	Get and set the clipping region.
	The **clipping region** is an arbitrary region used to limit drawing to any region within the port rectangle. The default clipping region is set arbitrarily large; however, your application can change this. At Fig 3, for example, `SetPortClipRegion` (or `ClipRect`) has been used to change Window B's clipping region so as to prevent the scroll bar areas being over-drawn.
	
	 WINDOW A
	WINDOW B
	TWO COLOUR GRAPHICS PORTS
	VISIBLE REGION OF WINDOW A
	MODIFIED CLIPPING REGION OF WINDOW B
	FIG 3 - VISIBLE REGION AND CLIPPING REGION

`GetPortForeColor` `RGBForeColor`	Get and set the foreground colour. These functions get and set an `RGBColor` structure that contains the *requested* **foreground colour**. By default, the foreground colour is black.
`GetPortBackColor` `RGBBackColor`	Get and set the background colour. These functions get and set an `RGBColor` structure that contains the *requested* **background colour**. By default, the backgroundground colour is white.

`GetPortBackPixPat`
`SetPortBackPixPat`
`BackPixPat`
`BackPat`
	Get and set the background pixel pattern.
	These functions get and set a handle to a `PixPat` structure (see below) that describes the background **pixel pattern**. Various QuickDraw functions use this pattern for filling scrolled or erased areas.

`GetPortPenPixPat`
`SetPortPenPixPat`
`PenPixPat`
`PenPat`
	Get and set the pen pixel pattern.
	These functions get and set a handle to a `PixPat` structure (see below) that describes the pixel pattern used by the graphics pen for drawing lines and framed shapes, and for painting shapes.

`GetPortFillPixPat`	Get the fill pixel pattern. This function gets a handle to a `PixPat` structure (see below) that describes the pixel pattern used when you call QuickDraw shape filling functions.
`GetPortPenLocation` `MoveTo`	Get and set the pen location. The pen location is the point where QuickDraw will begin drawing the next line, shape, or character. It can be anywhere on the coordinate plane.

`GetPortPenSize`
`SetPortPenSize`
`PenSize`
	Get and set the pen size.
	Pen size is the vertical height and horizontal width of the graphics pen. The default size is a 1-by-1 pixel square. If either the pen width or height is `0`, the pen does not draw.

`GetPortPenMode`
`SetPortPenMode`
`PenMode`
	Gets and sets the pen transfer mode.
	The pen **transfer mode** is a Boolean or arithmetic operation that determines how QuickDraw transfers the pen pattern to the pixel map during drawing operations. (See Chapter 12.)

`HidePen/ShowPen` `GetPortPenVisibility`	Gets and sets pen visibility. The pen's **visibility** means whether it draws on the screen.
`GetPortTextFont` `TextFont`	Get and set the font number for text. These functions get and set a **font family ID**, that is, anumber that identifies the font to be used in the graphics port.
`GetPortTextSize` `TextSize`	Get and set the text size. The text size is expressed in pixels, and is used by the Font Manager to provide the bitmaps for text drawing.
`GetPortTextFace` `TextFace`	Get and set the text style. The **style** of the text means, for example, bold, italic, and/or underlined.
`GetPortTextMode` `TextMode`	Get and set the text mode. The text mode is the transfer mode for text drawing, which functions much like the transfer mode specified in the `pnMode` field (see above).
`HiliteColor`	Get the highlight colour. (The highlight colour is copied to the graphics port from the low memory global `HiliteRGB`.)

You can open many graphics ports at the same time. Each has its own local coordinate system, drawing pattern, background pattern, pen size and location, foreground colour, background colour, pixel map, etc. You can instantly switch from one graphics port to another using the functions `SetPort`, `SetPortDialogPort`, and `SetPortWindowPort`.

When you use Window Manager and Dialog Manager functions to create windows, dialogs, and alerts, those managers automatically create graphics ports for you

Pixel Maps

QuickDraw draws in a **pixel map**. The graphics port object contains a handle to a pixel map, which is a data structure of type `PixMap`. A `PixMap` structure contains a pointer to a **pixel image**, as well as information on the image's storage format, depth, resolution, and colour usage. The `PixMap` structure is as follows:

```
struct PixMap
{
  Ptr        baseAddr;    // Pointer to image data.
  short      rowBytes;    // Flags, and bytes in a row.
  Rect       bounds;      // Boundary rectangle.
  short      pmVersion;   // Pixel Map version number.
  short      packType;    // Packing format.
  long       packSize;    // Size of data in packed state.
  Fixed      hRes;        // Horizontal resolution in dots per inch.
  Fixed      vRes;        // Vertical resolution in dots per inch.
  short      pixelType;   // Format of pixel image.
  short      pixelSize;   // Physical bits per pixel.
  short      cmpCount;    // Number of components in each pixel.
  short      cmpSize;     // Number of bits in each component.
  long       planeBytes;  // Offset to next plane.
  CTabHandle pmTable;     // Handle to a colour table for this image.
  long       pmReserved;  // (Reserved.)
};
typedef struct PixMap PixMap,*PixMapPtr,**PixMapHandle;
```

Field Descriptions

`baseAddr` In the case of an onscreen pixel image, a pointer to the first byte of the image data. Note that there can be several pixel maps pointing to the same pixel image, each imposing its own coordinate system on it.

A pixel image is analogous to the **bit image**. A bit image is a collection of bits in memory that form a grid. Fig 4 illustrates a bit image, which can be visualised as a matrix of rows and columns of bits with each row containing the same number of bytes. Each bit corresponds to one screen pixel. If a bit's value is 0, its screen pixel is white; if the bit's value is 1, the screen pixel is black. A pixel image is essentially the same as a bit image, except that a number of bits, not just one bit, are assigned to each pixel. The number of bits per pixel in a pixel image is called the pixel depth.

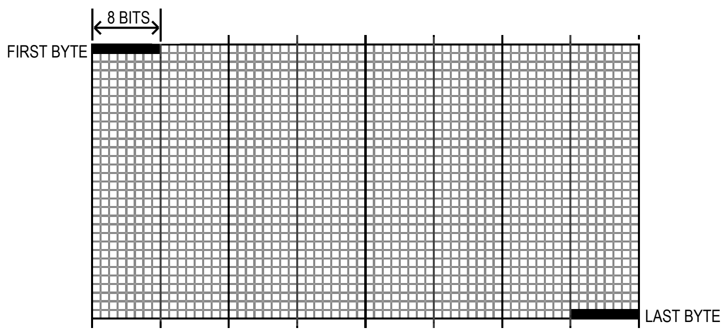


FIG 4 - A BIT IMAGE

`rowBytes` The offset in bytes from one row of the image to the next.

`bounds`

Mac OS 8/9

On Mac OS 8/9,the boundary rectangle defines the area of the pixel image into which QuickDraw can draw and provides the link between the local coordinate system of a graphics port and QuickDraw's global coordinate system. All drawing in a graphics port occurs in the intersection of the boundary rectangle and the port rectangle (and, within that intersection, all drawing is cropped to the graphics port's visible region and its clipping region).

As shown at Fig 5, on Mac OS 8/9, QuickDraw assigns the entire screen as the boundary rectangle. The boundary rectangle shares the same local coordinate system as the port rectangle of the window.

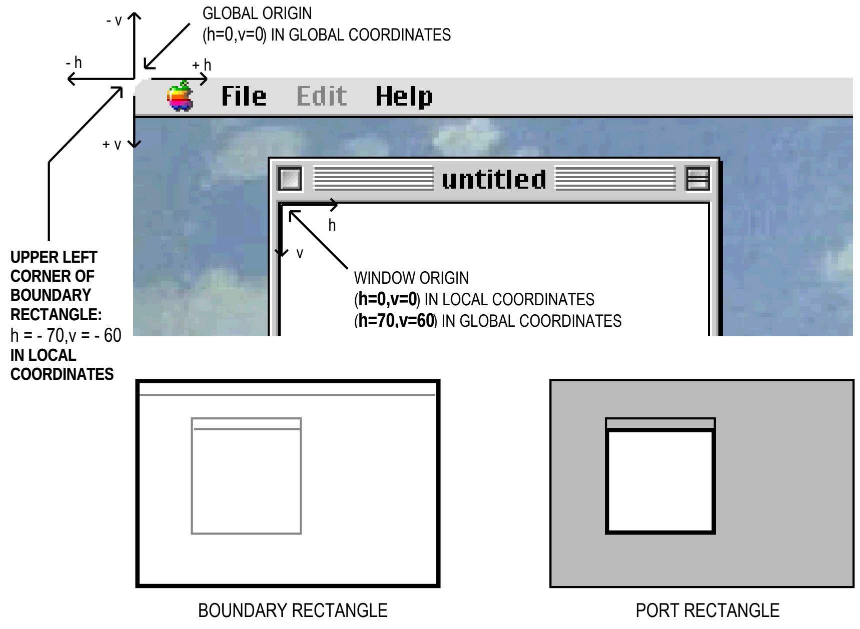

- v
GLOBAL ORIGIN
(h=0,v=0) IN GLOBAL COORDINATES
- h
+ h
File
Edit
Help
+ v
untitled
h
v
UPPER LEFT CORNER OF BOUNDARY RECTANGLE: h = - 70,v = - 60 IN LOCAL COORDINATES
WINDOW ORIGIN
(h=0,v=0) IN LOCAL COORDINATES
(h=70,v=60) IN GLOBAL COORDINATES
BOUNDARY RECTANGLE
PORT RECTANGLE

FIG 5 - LOCAL AND GLOBAL COORDINATE SYSTEMS, THE BOUNDARY RECTANGLE AND THE PORT RECTANGLE - MAC OS 8/9

You should not, incidentally, use the `bounds` field to determine the size of the screen; instead, use the `gdRect` field of the `GDevice` structure (see below).

Mac OS X

On Mac OS X, this field contains the bounds of the Core Graphics window that backs the Carbon window, and different mechanisms are employed to determine where the window's pixel map should be drawn.

`pmVersion` The QuickDraw version number that created this `PixMap` structure.

`packType` The packing algorithm used to compress the image data.

`packSize` The size of the packed image.

`hRes` The horizontal resolution of the pixel image in pixels per inch, abbreviated as dpi (dots per inch). By default, the value here is 0x00480000 (for 72 dpi), but QuickDraw supports `PixMap` structures of other resolutions. For example, `PixMap` structures for scanners can have dpi resolutions of 150, 200, 300, or greater.

`vRes` The vertical resolution. (See `hRes`).

`pixelType` The storage format. 0 indicates indexed pixels. 16 (`RGBDirect`) indicates direct pixels.

`pixelSize`	The number of bits used to represent a pixel.
`cmpCount`	The number of components used to represent a colour for a pixel. For indexed pixels, this field contains 1. For direct pixels this field contains the value 3.
`cmpSize`	The size of each colour component. For indexed devices, this is the same value as that in the `pixelSize` field. For direct devices, each of the three colour components can be either 5 bits for a 16-bit pixel (one of these 16 bits is unused), or 8 bits for a 32 bit pixel (8 of these 32 bits are unused). (See Translation of RGB Colours to Pixel Values, below.)
`planeBytes`	Multiple-plane images are not supported, so the value of this field is always 0.
`pmTable`	A handle to the `ColorTable` structure. `ColorTable` structures define the colours available for pixel images on indexed devices. Pixel images on direct devices do not need a colour table because the colours are stored right in the pixel values. In the case of direct devices, `pmTable` points to a dummy colour table.

Functions

Carbon introduced the following functions relating to pixel maps:

Function	Description
`GetPixBounds`	Get the pixel map's boundary rectangle.
`GetPixDepth`	Gets the pixel map's pixel depth.

Pixel Patterns and Bit Patterns

Pixel Patterns

The graphics port object stores handles to pixel patterns, structures of type `PixPat`.

Pixel patterns, which define a repeating design, can use colours at any pixel depth, and can be of any width and height that is a power of 2. You can create your own pixel patterns in your program code, but it is usually more convenient to store them in resources of type `'ppat'`. Fig 6 shows an 8-by-8 pixel `'ppat'` resource being created using Resorcerer.

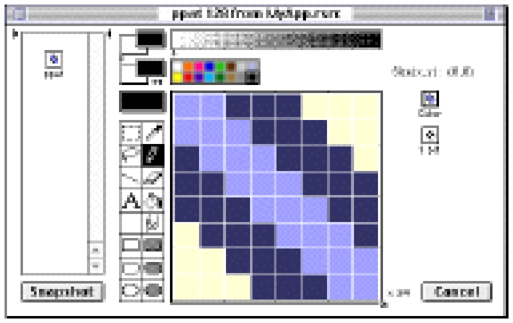
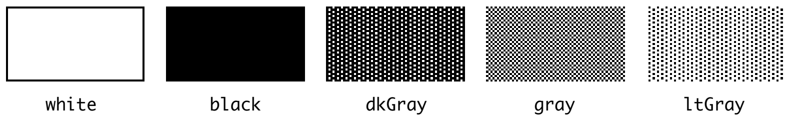


FIG 6 - CREATING A 'ppat' RESOURCE USING RESORCERER

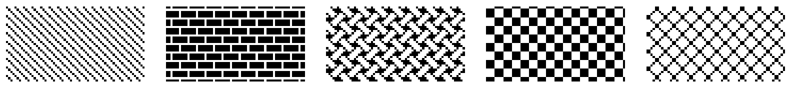
Bit Patterns

Bit patterns date from the era of the black-and-white Macintosh, but may be stored in a colour graphics port object. (`PixPat` structures can contain bit patterns as well as pixel patterns.) Bit patterns are defined in data structures of type `Pattern`, a 64-pixel image of a repeating design organised as an 8-by-8 pixel square.

Five bit patterns are pre-defined as QuickDraw global variables. The five pre-defined patterns are available not only through the QuickDraw globals but also as system resources. Fig 7 shows images drawn using some of the 38 available system-supplied bit patterns.



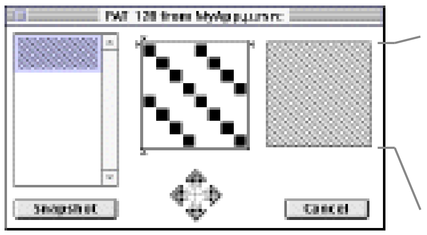
RECTANGLES DRAWN WITH BIT PATTERNS PRE-DEFINED AS QUICKDRAW GLOBAL VARIABLES



RECTANGLES DRAWN WITH OTHER BIT PATTERNS IN THE SYTEM RESOURCE FILE

FIG 7 - RECTANGLES DRAWN USING BIT PATTERNS IN THE SYSTEM RESOURCE FILE

You can create your own bit patterns in your program code, but it is usually more convenient to store them in resources of type `'PAT '` or `'PAT#'`. Fig 8 shows a `'PAT '` resource being created using Resorcerer, together with the contents of the `pat` field of the structure of type `Pattern` that is created when the resource is loaded.



```
pat[0] = 10001000 = 0x88
pat[1] = 01000100 = 0x44
pat[2] = 00100010 = 0x22
pat[3] = 00010001 = 0x11
pat[4] = 10001000 = 0x88
pat[5] = 01000100 = 0x44
pat[6] = 00100010 = 0x22
pat[7] = 00010001 = 0x11
```

FIG 8 - CREATING A 'PAT ' RESOURCE USING RESORCERER

Creating Graphics Ports

Your application creates a (colour) graphics port using either the `GetNewCWindow`, `NewCWindow`, or `NewGWorld` function. These functions automatically call `CreateNewPort`, which opens the port.

Translation of RGB Colours to Pixel Values

As previously stated, the graphics port object contains a pointer to the beginning of the onscreen pixel image. When your application specifies an RGB colour for a pixel in the pixel image, QuickDraw translates that colour into a value appropriate for display on the user's screen. QuickDraw stores this value in the pixel. The **pixel value** is a number used by system software and a graphics device to represent a colour. The translation from the colour you specify in an `RGBColor` structure to a pixel value is performed at the time you draw the colour. The process differs for direct and indexed devices as follows:

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

Your application never needs to handle pixel values. However, to clarify the relationship between `RGBColor` structures and the pixels that are actually displayed, the following presents some examples of the derivation of pixel values from `RGBColor` structures.

Derivation of Pixel Values on Indexed Devices

Fig 9 shows the translation of an `RGBColor` structure to an 8-bit pixel value on an indexed device.

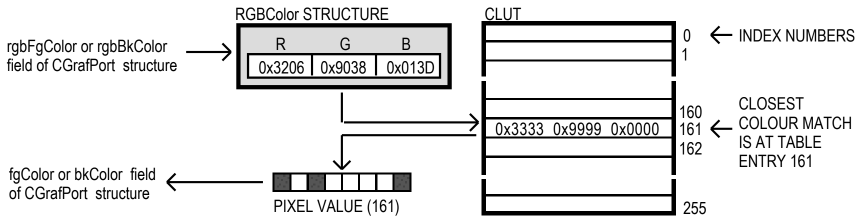

RGBColor STRUCTURE
CLUT
rgbFgColor or rgbBkColor field of CGrafPort structure
R G B
0x3206 0x9038 0x013D
0 ← INDEX NUMBERS
1
160
0x3333 0x9999 0x0000 161 ← CLOSEST COLOUR MATCH IS AT TABLE ENTRY 161
162
fgColor or bkColor field of CGrafPort structure
PIXEL VALUE (161)
255

FIG 9 - TRANSLATING AN RGBColor STRUCTURE TO AN 8-BIT PIXEL VALUE ON AN INDEXED DEVICE

An application might call `GetCPixel` to determine the colour of a pixel set by the pixel value at Fig 9. As shown at Fig 10, the Color Manager uses the pixel value (an index number) to find the `RGBColor` structure stored in the CLUT for that pixel's colour. This is the colour returned by `GetCPixel`. As shown at Fig 10, this is not necessarily the exact colour first specified.

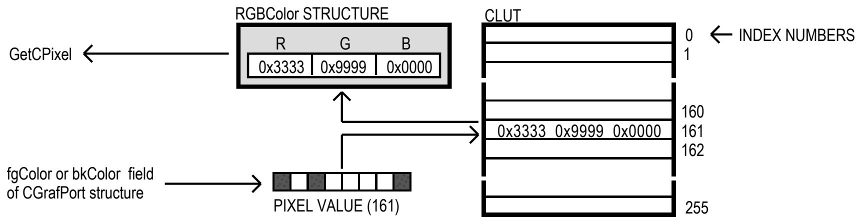


FIG 10 - TRANSLATING AN 8-BIT PIXEL VALUE ON AN IDEXED DEVICE TO AN RGBColor STRUCTURE

Derivation of Pixel Values on Direct Devices

Fig 11 shows how QuickDraw converts an `RBGColor` structure into a 16-bit pixel value on a direct device. The most significant 5 bits of each field of the `RGBColor` structure are stored in the lower 15 bits of the pixel value. The high bit is unused. Fig 11 also shows how QuickDraw expands a 16-bit pixel value to a 48-bit `RGBColor` structure. Each 5-bit component, and the most significant bit, are inserted into each 16-bit field of the `RGBColor` structure. Note the difference between the result and the original 48-bit value.

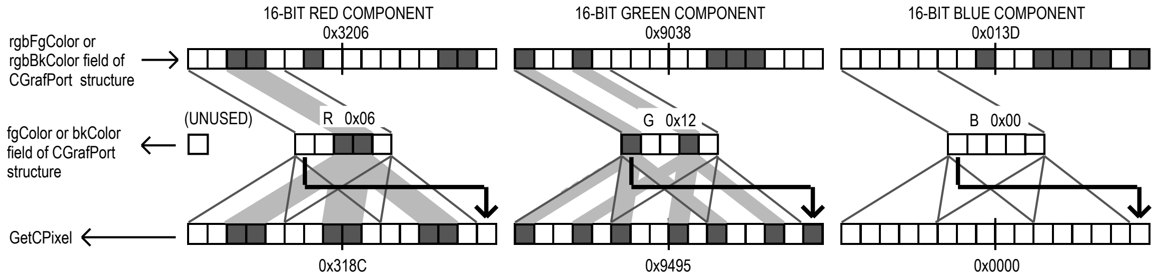


FIG 11 - TRANSLATING AN RGBColor STRUCTURE TO A 16 BIT PIXEL VALUE, AND FROM A 16-BIT PIXEL VALUE TO AN RGBColor STRUCTURE, ON A DIRECT DEVICE

Fig 12 shows how QuickDraw converts an `RBGColor` structure into a 32-bit pixel value on a direct device. The most significant 8 bits of each 16-bit field of the `RGBColor` structure are stored in the lower 3 bytes of the pixel value. 8 bits in the high byte of the pixel value are unused. Fig 12 also shows how QuickDraw expands a 32-bit pixel value to an `RBGColor` structure. Each of the 8-bit components is doubled. Note the difference between the result and the original 48-bit value.

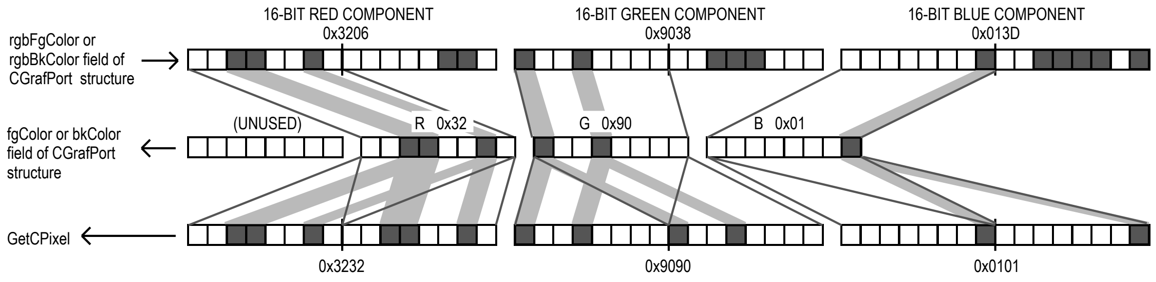


FIG 12 - TRANSLATING AN RGBColor STRUCTURE TO A 32 BIT PIXEL VALUE, AND FROM A 32-BIT PIXEL VALUE TO AN RGBColor STRUCTURE, ON A DIRECT DEVICE

Colours on Grayscale Screens

When QuickDraw displays a colour on a grayscale screen, it computes the luminance, or intensity of light, of the desired colour and uses that value to determine the appropriate gray value to draw.

A grayscale device can be a graphics device that the user sets to grayscale. For such a graphics device, Colour QuickDraw places an evenly spaced set of grays in the graphics device's CLUT.

Graphics Devices and GDevice Structures

As previously stated, QuickDraw provides a device-independent interface. Your application can draw images in the graphics port for a window and QuickDraw automatically manages the path to the screen — even if the user is using multiple screens. QuickDraw communicates with a video device, such as a plug-in video card or a built-in video interface, by automatically creating and managing a structure of type `GDevice`.

Types of Graphics Device

A **graphics device** is anything QuickDraw can draw into. There are three types of graphics device:

- **Video devices**, which control screens.
- **Offscreen graphics worlds**. (See Chapter 13.)
- **Printing graphics ports**. (See Chapter 15.)

In the case of a video device for an offscreen graphics world, QuickDraw automatically creates, and stores state information in, a `GDevice` structure.

GDevice Structure

QuickDraw creates and initialises a `GDevice` structure for each video device found during startup. QuickDraw also automatically creates a `GDevice` structure when you call `NewGWorld` to create an offscreen graphics world.

A list called a **device list** links together all existing `GDevice` structures. The **current device**, which is sometimes called the **active device**, is that device in the device list into which drawing is currently taking place.

Your application generally never needs to create `GDevice` structures; however, in may need to examine `GDevice` structures to determine the capabilities of the user's screens. The `GDevice` structure is as follows:

```
struct GDevice
{
  short        gdRefNum;      // Reference Number of Driver.
  short        gdID;          // Client ID for search procedures.
  short        gdType;        // Type of device (indexed or direct).
  ITabHandle   gdITable;      // Handle to inverse lookup table for Color Manager.
  short        gdResPref;     // Preferred resolution.
  SProcHndl    gdSearchProc;  // Handle to list of search functions.
```

```
    CProcHndl    gdCompProc;    // Handle to list of complement functions.
    short        gdFlags;       // Graphics device flags.
    PixMapHandle gdPMap;        // Handle to pixel map for displayed image.
    long         gdRefCon;      // Reference value.
    Handle       gdNextGD;      // Handle to next GDevice structure.
    Rect         gdRect;        // Device's global boundaries.
    long         gdMode;        // Device's current mode.
    short        gdCCBytes;     // Width of expanded cursor data.
    short        gdCCDepth;     // Depth of expanded cursor data.
    Handle       gdCCXData;     // Handle to cursor's expanded data.
    Handle       gdCCXMask;     // Handle to cursor's expanded mask.
    long         gdReserved;    // (Reserved.  Must be 0.)
  };
  typedef struct GDevice GDevice;
  typedef GDevice *GDPtr, **GDHandle;
```

Main Field Descriptions

`gdType` The type of graphics device. The flag bits of this field are as follows:

Constant	Bit	Meaning If Set
`clutType`	`0`	CLUT device.
`fixedType`	`1`	Fixed CLUT device.
`directType`	`2`	Direct device.

`gdITable` Points to an **inverse table**. This is a special Color Manager data structure that allows index numbers in a CLUT to be found very quickly.

`gdFlags` Device attributes (that is, whether the device is a screen, whether it is the main screen, whether it is set to black-and-white or colour, whether it is the active device, etc.). The main flag bits in this field are as follows:

Constant	Bit	Meaning If Set
`gdDevType`	`0`	Device is a colour device. (If not set, device is a black-and-white divice.)
`mainScreen`	`11`	Device is the main screen.
`screenDevice`	`13`	Device is a screen device.
`screenActive`	`15`	Device is current device.

`gdPMap` A handle to the pixel map (`PixMap`) structure.

`gdNextGD` A handle to the next device in the device list. Contains 0 if this is the last graphics device in the device list.

`gdRect` The boundary rectangle of this graphics device. The upper-left corner of the boundary rectangle for the main screen is set to (`0,0`) and all other graphics devices are relative to this.

Setting a Device's Pixel Depth

The `gdPMap` field of the `GDevice` structure contains a handle to a `PixMap` structure which, in turn, contains the `PixelSize` field to which is assigned the pixel depth of the device.

The user can change the pixel depth of video devices. Accordingly, although your application may have a preferred pixel depth, it should be flexible enough to accommodate other pixel depths.

Your application can change the pixel depth using `SetDepth`. However, before calling this function, you should call the `HasDepth` function to confirm that the hardware can support the desired pixel depth. Generally speaking, you should not change pixel depth without first seeking the consent of the user via an alert or dialog.

Other Graphics Managers

In addition to the QuickDraw functions, several other collections of system software functions are available to assist you in drawing images.

Palette Manager

Your application can use the Palette Manager to provide more sophisticated colour support on indexed graphics devices. The Palette Manager allows your application to specify sets of colours that it needs on a window-by-window basis.

Color Picker Utilities

To solicit colour choices from users, your application can use the Color Picker Utilities. The Color Picker Utilities also provide functions that allow your application to convert between colours specified in `RGBColor` structures and colours specified for other colour models, such as the CMYK (cyan, magenta, yellow, black) model used for many colour printers. (See Chapter 25.)

Coping With Multiple Monitors

Aspects of coping with a multiple monitors environment are addressed at Chapter 25.

Relevant QuickDraw Constants, Data Types, and Functions

Constants

Flag Bits of gdType Field of GDevice Structure

```
clutType     = 0
fixedType    = 1
directType   = 2
```

Flag Bits of gdFlags Field of GDevice Structure

```
gdDevType    = 0
burstDevice  = 7
ext32Device  = 8
ramInit      = 10
mainScreen   = 11
allInit      = 12
screenDevice = 13
noDriver     = 14
screenActive = 15
```

Pixel Type

```
RGBDirect     = 16  16 and 32 bits-per-pixel pixelType value.
```

Data Types

```
typedef struct OpaqueGrafPtr* GrafPtr;
typedef GrafPtr CGrafPtr;
```

Pixel Map

```
struct PixMap
{
  Ptr          baseAddr;     // Pointer to image data.
  short        rowBytes;     // Flags, and bytes in a row.
  Rect         bounds;       // Boundary rectangle.
  short        pmVersion;    // Pixel Map version number.
  short        packType;     // Packing format.
  long         packSize;     // Size of data in packed state.
  Fixed        hRes;         // Horizontal resolution in dots per inch.
  Fixed        vRes;         // Vertical resolution in dots per inch.
  short        pixelType;    // Format of pixel image.
  short        pixelSize;    // Physical bits per pixel.
  short        cmpCount;     // Number of components in each pixel.
  short        cmpSize;      // Number of bits in each component.
  long         planeBytes;   // Offset to next plane.
  CTabHandle pmTable;        // Handle to a colour table for this image.
  long  pmReserved;          // (Reserved.)
};
typedef struct PixMap PixMap,*PixMapPtr,**PixMapHandle;
```

BitMap

```
struct BitMap
{
  Ptr          baseAddr;      // Pointer to bit image.
  short        rowBytes;      // Row width.
  Rect         bounds;        // Boundary rectangle.
};
typedef struct BitMap BitMap;
typedef BitMap *BitMapPtr, **BitMapHandle;
```

Pixel Pattern

```
struct PixPat
{
  short        patType;       // Type of pattern.
  PixMapHandle patMap;        // The pattern's pixel map.
  Handle       patData;       // Pixel map's data.
  Handle       patXData;      // Expanded Pattern data (internal use).
  short        patXValid;     // Flags whether expanded Pattern valid.
  Handle       patXMap;       // Handle to expanded Pattern data (reserved).
  Pattern      pat1Data;      // Bit map's data.
};
typedef struct PixPat PixPat;
typedef PixPat *PixPatPtr;
typedef PixPatPtr *PixPatHandle;
```

Pattern

```
struct Pattern
{
  UInt8        pat[8];
};
typedef struct Pattern Pattern;
typedef Pattern *PatPtr;
typedef PatPtr *PatHandle;
```

GDevice

```
struct GDevice
{
  short        gdRefNum;      // Reference Number of Driver.
  short        gdID;          // Client ID for search procedures.
  short        gdType;        // Type of device (indexed or direct).
  ITabHandle   gdITable;      // Handle to inverse lookup table for Color Manager.
  short        gdResPref;     // Preferred resolution.
  SProcHndl    gdSearchProc;  // Handle to list of search functions.
  CProcHndl    gdCompProc;    // Handle to list of complement functions.
  short        gdFlags;       // Graphics device flags.
  PixMapHandle gdPMap;        // Handle to pixel map for displayed image.
  long         gdRefCon;      // Reference value.
  Handle       gdNextGD;      // Handle to next GDevice structure.
  Rect         gdRect;        // Device's global boundaries.
  long         gdMode;        // Device's current mode.
  short        gdCCBytes;     // Width of expanded cursor data.
  short        gdCCDepth;     // Depth of expanded cursor data.
  Handle       gdCCXData;     // Handle to cursor's expanded data.
  Handle       gdCCXMask;     // Handle to cursor's expanded mask.
  long         gdReserved;    // (Reserved.  Must be 0.)
};
typedef struct GDevice GDevice;
typedef GDevice *GDPtr, **GDHandle;
```

Functions

Opening and Closing Graphics Ports

```
CgrafPtr      CreateNewPort(void);
Void          DisposePort(CGrafPtr port);
```

Saving and Restoring Graphics Ports

```
void          GetPort(GrafPtr *port);
void          SetPort(GrafPtr port);
void          SetPortDialogPort(DialogPtr dialog);
void          SetPortWindowPort(WindowRef window);
```

Getting a Pointer to the Owning Window

```
WindowRef     GetWindowFromPort(CGrafPtr port);
```

Graphics Port Accessors

```
PixMapHandle  GetPortPixMap(CGrafPtr port);
Rect          GetPortBounds(CGrafPtr port,Rect *rect);
void          SetPortBounds(CGrafPtr port,const Rect *rect);
RgnHandle     GetPortVisibleRegion(CGrafPtr port,RgnHandle visRgn);
void          SetPortVisibleRegion(CGrafPtr port,RgnHandle visRgn);
RgnHandle     GetPortClipRegion(CGrafPtr port,RgnHandle clipRgn);
void          SetPortClipRegion(CGrafPtr port,RgnHandle clipRgn);
void          SetClip(RgnHandle rgn);
```

```
RGBColor      GetPortForeColor(CGrafPtr port,RGBColor *foreColor);
void          RGBForeColor(const RGBColor *color);
RGBColor      GetPortBackColor(CGrafPtr port,RGBColor *backColor);
void          RGBBackColor(const RGBColor *color);
PixPatHandle  GetPortBackPixPat(CGrafPtr port,PixPatHandle backPattern);
void          SetPortBackPixPat(CGrafPtr port,PixPatHandle backPattern);
void          BackPat(const Pattern *pat);
PixPatHandle  GetPortPenPixPat(CGrafPtr port,PixPatHandle penPattern);
void          SetPortPenPixPat(CGrafPtr port,PixPatHandle penPattern);
void          PenPat(const Pattern *pat);
PixPatHandle  GetPortFillPixPat(CGrafPtr port,PixPatHandle fillPattern);
Point         GetPortPenLocation(CGrafPtr port,Point *penLocation);
void          MoveTo(short h,short v);
Point         GetPortPenSize(CGrafPtr port,Point *penSize);
void          SetPortPenSize(CGrafPtr port,Point penSize);
void          PenSize(short width,short height);
SInt32        GetPortPenMode(CGrafPtr port);
void          SetPortPenMode(CGrafPtr port,SInt32 penMode);
void          PenMode(short mode);
short         GetPortTextFont(CGrafPtr port);
void          TextFont(short font);
void          HidePen(void);
void          ShowPen(void);
short         GetPortPenVisibility(CGrafPtr port);
short         GetPortTextSize(CGrafPtr port);
void          TextSize(short size);
Style         GetPortTextFace(CGrafPtr port);
void          TextFace(StyleParameter face);
short         GetPortTextMode(CGrafPtr port);
void          TextMode(short mode)
RGBColor      GetPortHiliteColor(CGrafPtr port,RGBColor *hiliteColor);
```

Creating, Setting, Disposing of, and Accessing Pixel Maps

```
PixMapHandle  NewPixMap(void);
void          CopyPixMap(PixMapHandle srcPM,PixMapHandle dstPM);
void          SetPortPix(PixMapHandle pm);
void          DisposePixMap(PixMapHandle pm);
Rect          GetPixBounds(PixMapHandle pixMap,Rect *bounds);
short         GetPixDepth(PixMapHandle pixMap);
```

Creating, Setting and Disposing of Graphics Device Structures

```
GDHandle      NewGDevice(short refNum,long mode);
void          InitGDevice(short qdRefNum,long mode,GDHandle gdh);
void          SetDeviceAttribute(GDHandle gdh,short attribute,Boolean value);
void          SetGDevice(GDHandle gd);
void          DisposeGDevice(GDHandle gdh);
```

Getting the Available Graphics Devices

```
GDHandle      GetGDevice(void);
GDHandle      GetMainDevice(void);
GDHandle      GetNextDevice(GDHandle curDevice);
GDHandle      GetDeviceList(void);
```

Determining the Characteristics of a Video Device

```
Boolean       TestDeviceAttribute(GDHandle gdh,short attribute);
void          ScreenRes(short *scrnHRes,short *scrnVRes);
```

Changing the Pixel Depth of a Video Device

```
OSErr         SetDepth(GDHandle gd,short depth,short whichFlags,short flags);
short         HasDepth(GDHandle gd,short depth,short whichFlags,short flags);
```

Demonstration Program PreQuickDraw Listing

```
// *******************************************************************************************
// PreQuickDraw.c                                                         CLASSIC EVENT MODEL
// *******************************************************************************************
//
// This program opens a window in which is displayed some information retrieved from the
// GDevice structure for the main video device, from the graphics port's pixel map, and from
// the graphics port object using QuickDraw functions.
//
// A Demonstration menu allows the user to set the monitor to various pixel depths and to
// restore the original pixel depth.  Setting the monitor to a pixel depth of 8 (256 colours)
// or less causes the colours in the colour table to be displayed.
//
// The program utilises 'plst', 'MBAR', 'MENU', 'WIND', and 'STR#' resources, and a 'SIZE'
// resource with the acceptSuspendResumeEvents, canBackground, doesActivateOnFGSwitch, and
// isHighLevelEventAware flags set.
//
// *******************************************************************************************

// ………………………………………………………………………………………………………………………………………………………………………………… includes

#include <Carbon.h>

// ………………………………………………………………………………………………………………………………………………………………………………… defines

#define rMenubar              128
#define rWindow               128
#define mAppleApplication     128
#define  iAbout               1
#define mFile                 129
#define  iQuit                12
#define mDemonstration        131
#define  iSetDepth8           1
#define  iSetDepth16          2
#define  iSetDepth32          3
#define  iRestoreStartDepth   5
#define rIndexedStrings       128
#define  sMonitorInadequate   1
#define  sMonitorAtThatDepth  2
#define  sMonitorAtStartDepth 3
#define  sRestoringMonitor    4
#define MAX_UINT32            0xFFFFFFFF

// ……………………………………………………………………………………………………………………………………………………………………… global variables

Boolean gDone;
SInt16  gStartupPixelDepth;

// …………………………………………………………………………………………………………………………………………………………………… function prototypes

void    main                        (void);
void    doPreliminaries             (void);
OSErr   quitAppEventHandler         (AppleEvent *,AppleEvent *,SInt32);
void    doEvents                    (EventRecord *);
void    doDisplayInformation        (WindowRef);
Boolean doCheckMonitor              (void);
void    doSetMonitorPixelDepth      (SInt16);
void    doRestoreMonitorPixelDepth  (void);
void    doMonitorAlert              (Str255);

// ************************************************************************************** main

void  main(void)
{
  MenuBarHandle menubarHdl;
  SInt32        response;
  MenuRef       menuRef;
  WindowRef     windowRef;
  SInt16        entries = 0;
  Str255        theString;
  EventRecord   EventStructure;

  // .................................................................... do preliminaries

  doPreliminaries();

  // ............................................................. set up menu bar and menus

  menubarHdl = GetNewMBar(rMenubar);
  if(menubarHdl == NULL)
    ExitToShell();
  SetMenuBar(menubarHdl);
  DrawMenuBar();

  Gestalt(gestaltMenuMgrAttr,&response);
  if(response & gestaltMenuMgrAquaLayoutMask)
  {
    menuRef = GetMenuRef(mFile);
    if(menuRef != NULL)
    {
      DeleteMenuItem(menuRef,iQuit);
      DeleteMenuItem(menuRef,iQuit - 1);
      DisableMenuItem(menuRef,0);
    }
  }

  // ......................................... check if monitor can display at least 16-bit colour

  if(!doCheckMonitor())
  {
    GetIndString(theString,rIndexedStrings,sMonitorInadequate);
    doMonitorAlert(theString);
  }

  // ......................................... open windows, set font, show windows, move windows

  if(!(windowRef = GetNewCWindow(rWindow,NULL,(WindowRef)-1)))
    ExitToShell();

  SetPortWindowPort(windowRef);
  TextSize(10);

  // ...................................................................... enter eventLoop

  gDone = false;

  while(!gDone)
  {
    if(WaitNextEvent(everyEvent,&EventStructure,MAX_UINT32,NULL))
      doEvents(&EventStructure);
  }
}

// *************************************************************************** doPreliminaries

void  doPreliminaries(void)
{
  OSErr osError;

  MoreMasterPointers(32);
  InitCursor();
  FlushEvents(everyEvent,0);

  osError = AEInstallEventHandler(kCoreEventClass,kAEQuitApplication,
                          NewAEEventHandlerUPP((AEEventHandlerProcPtr) quitAppEventHandler),
                          0L,false);
  if(osError != noErr)
    ExitToShell();
}

// ************************************************************************** doQuitAppEvent

OSErr  quitAppEventHandler(AppleEvent *appEvent,AppleEvent *reply,SInt32 handlerRefcon)
{
  OSErr    osError;
  DescType returnedType;
  Size     actualSize;

  osError = AEGetAttributePtr(appEvent,keyMissedKeywordAttr,typeWildCard,&returnedType,NULL,0,
                              &actualSize);

  if(osError == errAEDescNotFound)
  {
    gDone = true;
    osError = noErr;
  }
  else if(osError == noErr)
    osError = errAEParamMissed;

  return osError;
}

// ******************************************************************************** doEvents

void  doEvents(EventRecord *eventStrucPtr)
{
  SInt32         menuChoice;
  MenuID         menuID;
  MenuItemIndex  menuItem;
  WindowPartCode partCode;
  WindowRef      windowRef;
  Rect           portRect;

  switch(eventStrucPtr->what)
  {
    case kHighLevelEvent:
      AEProcessAppleEvent(eventStrucPtr);
      break;

    case keyDown:
      if((eventStrucPtr->modifiers & cmdKey) != 0)
      {
        menuChoice = MenuEvent(eventStrucPtr);
        menuID = HiWord(menuChoice);
        menuItem = LoWord(menuChoice);
        if(menuID == mFile && menuItem  == iQuit)
          gDone = true;
      }
      break;

    case mouseDown:
      if(partCode = FindWindow(eventStrucPtr->where,&windowRef))
      {
        switch(partCode)
        {
          case inMenuBar:
            menuChoice = MenuSelect(eventStrucPtr->where);
            menuID = HiWord(menuChoice);
            menuItem = LoWord(menuChoice);

            if(menuID == 0)
              return;

            switch(menuID)
            {
              case mAppleApplication:
                if(menuItem == iAbout)
                  SysBeep(10);
                break;

              case mFile:
                if(menuItem == iQuit)
                  gDone = true;
                break;

              case mDemonstration:
                if(menuItem == iSetDepth8)
                  doSetMonitorPixelDepth(8);
                else if(menuItem == iSetDepth16)
                  doSetMonitorPixelDepth(16);
                else if(menuItem == iSetDepth32)
                  doSetMonitorPixelDepth(32);
                else if(menuItem == iRestoreStartDepth)
                  doRestoreMonitorPixelDepth();
                break;
            }
            HiliteMenu(0);
            break;

          case inDrag:
            DragWindow(windowRef,eventStrucPtr->where,NULL);
            GetWindowPortBounds(windowRef,&portRect);
            InvalWindowRect(windowRef,&portRect);
            break;
        }
      }
      break;

    case updateEvt:
      windowRef = (WindowRef) eventStrucPtr->message;
      BeginUpdate(windowRef);
      SetPortWindowPort(windowRef);
      doDisplayInformation(windowRef);
      EndUpdate(windowRef);
      break;
  }
}

// ******************************************************************** doDisplayInformation

void  doDisplayInformation(WindowRef windowRef)
{
  RGBColor     whiteColour = { 0xFFFF, 0xFFFF, 0xFFFF };
  RGBColor     blueColour  = { 0x3333, 0x3333, 0x9999 };
  Rect         portRect;
  GDHandle     deviceHdl;
  SInt16       videoDeviceCount = 0;
  Str255       theString;
  SInt16       deviceType, pixelDepth, bytesPerRow;
  Rect         theRect;
  GrafPtr      grafPort;
  PixMapHandle pixMapHdl;
  CTabHandle   colorTableHdl;
  SInt16       entries = 0, vert = 28, horiz = 250, index = 0;
  RGBColor     getPixelColour,colourTableColour;

  RGBForeColor(&whiteColour);
  RGBBackColor(&blueColour);
  GetWindowPortBounds(windowRef,&portRect);
  EraseRect(&portRect);
  QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);

  // .......................................................................... Get Device List
```

```
deviceHdl = GetDeviceList();

// ...................................................... count video devices in device list

while(deviceHdl != NULL)
{
  if(TestDeviceAttribute(deviceHdl,screenDevice))
    videoDeviceCount ++;

  deviceHdl = GetNextDevice(deviceHdl);
}

NumToString(videoDeviceCount,theString);
MoveTo(10,20);
DrawString(theString);
if(videoDeviceCount < 2)
  DrawString("\p video device in the device list.");
else
  DrawString("\p video devices in the device list.");

// ........................................................................... Get Main Device

deviceHdl = GetMainDevice();

// ................................................................... determine device type

MoveTo(10,35);

if(((1 << gdDevType) & (*deviceHdl)->gdFlags) != 0)
  DrawString("\pThe main video device is a colour device.");
else
  DrawString("\pThe main video device is a monochrome device.");

MoveTo(10,50);
deviceType = (*deviceHdl)->gdType;
switch(deviceType)
{
  case clutType:
    DrawString("\pIt is an indexed device with variable CLUT.");
    break;

  case fixedType:
    DrawString("\pIt is is an indexed device with fixed CLUT.");
    break;

  case directType:
    DrawString("\pIt is a direct device.");
    break;
}

// ................................................................... Get Handle to Pixel Map

grafPort = GetWindowPort(windowRef);
pixMapHdl = GetPortPixMap(grafPort);
// pixMapHdl = (*deviceHdl)->gdPMap; // alternative method

// ............................................................. get and display pixel depth

MoveTo(10,70);
DrawString("\pPixel depth = ");

pixelDepth = GetPixDepth(pixMapHdl);
// pixelDepth = (*(*deviceHdl)->gdPMap)->pixelSize;  // alternative method

NumToString(pixelDepth,theString);
DrawString(theString);

// ........................................................... get and display bytes per row
```

```
MoveTo(10,90);
bytesPerRow = (*pixMapHdl)->rowBytes & 0x7FFF;
DrawString("\pBytes per row = ");
NumToString(bytesPerRow,theString);
DrawString(theString);

// ………………………………………………………………………………………………………………… Get Device's Global Boundary Rectangle

theRect = (*deviceHdl)->gdRect;

// ......................................... calculate and display total pixel image bytes

MoveTo(10,105);
DrawString("\pTotal pixel image bytes = ");
NumToString(bytesPerRow * theRect.bottom,theString);
DrawString(theString);

// ................................................... display device's boundary rectangle

MoveTo(10,130);
TextFace(bold);
DrawString("\pGraphics Device's Boundary Rectangle");
TextFace(normal);
MoveTo(10,145);
DrawString("\p(gdRect field of GDevice structure)");

MoveTo(10,160);
DrawString("\pBoundary rectangle top = ");
NumToString(theRect.top,theString);
DrawString(theString);

MoveTo(10,175);
DrawString("\pBoundary rectangle left = ");
NumToString(theRect.left,theString);
DrawString(theString);

MoveTo(10,190);
DrawString("\pBoundary rectangle bottom = ");
NumToString(theRect.bottom,theString);
DrawString(theString);

MoveTo(10,205);
DrawString("\pBoundary rectangle right = ");
NumToString(theRect.right,theString);
DrawString(theString);

// ……………………………………………………………………………………………… Get and Display Pixel Map's Boundary Rectangle

GetPixBounds(pixMapHdl,&theRect);

MoveTo(10,225);
TextFace(bold);
DrawString("\pPixel Map's Boundary Rectangle");
TextFace(normal);
MoveTo(10,240);
DrawString("\p(bounds field of PixMap structure)");

MoveTo(10,255);
DrawString("\pBoundary rectangle top = ");
NumToString(theRect.top,theString);
DrawString(theString);

MoveTo(10,270);
DrawString("\pBoundary rectangle left = ");
NumToString(theRect.left,theString);
DrawString(theString);

MoveTo(10,285);
DrawString("\pBoundary rectangle bottom = ");
```

```
NumToString(theRect.bottom,theString);
DrawString(theString);

MoveTo(10,300);
DrawString("\pBoundary rectangle right = ");
NumToString(theRect.right,theString);
DrawString(theString);

MoveTo(10,320);
DrawString("\pOn Mac OS X, drag window after pixel depth and screen resolution changes to");
DrawString("\p ensure that");
MoveTo(10,333);
DrawString("\pbytes per row, pixel image bytes, and colour values are updated.");

// ………………………………………………………………… Get and Display RGB Components of Requested Background Colour

MoveTo(250,255);
GetBackColor(&blueColour);
DrawString("\pRequested background colour (rgb) = ");
MoveTo(250,270);
NumToString(blueColour.red,theString);
DrawString(theString);
DrawString("\p  ");
NumToString(blueColour.green,theString);
DrawString(theString);
DrawString("\p  ");
NumToString(blueColour.blue,theString);
DrawString(theString);

// …………………… If Direct Device, Get and Display RGB Components of Colour Returned by GetCPixel

if(deviceType == directType)
{
  MoveTo(250,285);
  GetCPixel(10,10,&getPixelColour);
  DrawString("\pColour returned by CetCPixel (rgb) = ");
  MoveTo(250,300);
  NumToString(getPixelColour.red,theString);
  DrawString(theString);
  DrawString("\p  ");
  NumToString(getPixelColour.green,theString);
  DrawString(theString);
  DrawString("\p  ");
  NumToString(getPixelColour.blue,theString);
  DrawString(theString);
}

// ............................................ else prepare to display colour table index

else
{
  MoveTo(250,285);
  DrawString("\pBackground colour (colour table index):");
}

// ……………………………………………………………………………………………………………………………………… Get Handle to Colour Table

colorTableHdl = (*pixMapHdl)->pmTable;

// ...................................... if any entries in colour table, draw the colours

MoveTo(250,20);
DrawString("\pColour table:");

entries = (*colorTableHdl)->ctSize;

if(entries < 2)
{
  MoveTo(260,100);
```

```
    DrawString("\pOnly one (dummy) entry in the colour");
    MoveTo(260,115);
    DrawString("\ptable.  To cause the colour table to be");
    MoveTo(260,130);
    DrawString("\pbuilt, set the monitor to bit depth 8");
    MoveTo(260,145);
    DrawString("\p(256 colours), causing it to act like ");
    MoveTo(260,160);
    DrawString("\pan indexed device.");
    SetRect(&theRect,250,28,458,236);
    FrameRect(&theRect);
  }

  for(index = 0;index <= entries;index++)
  {
    SetRect(&theRect,horiz,vert,horiz+12,vert+12);
    colourTableColour = (*colorTableHdl)->ctTable[index].rgb;
    RGBForeColor(&colourTableColour);
    PaintRect(&theRect);

    // .... also, if device is not a  direct device, and current colour matches background ...

    if(deviceType == clutType || deviceType == fixedType)
    {
      if(colourTableColour.red == blueColour.red &&
         colourTableColour.green == blueColour.green &&
         colourTableColour.blue == blueColour.blue)
      {

        // ...................... outline the drawn colour and display the colour table index

        RGBForeColor(&whiteColour);
        InsetRect(&theRect,-1,-1);
        FrameRect(&theRect);
        MoveTo(250,300);
        NumToString(index,theString);
        DrawString(theString);
      }
    }

    horiz += 13;
    if(horiz > 445)
    {
      horiz = 250;
      vert += 13;
    }
  }

  QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
}

// ************************************************************************** doCheckMonitor

Boolean doCheckMonitor(void)
{
  GDHandle  mainDeviceHdl;

  mainDeviceHdl = GetMainDevice();

  if(!(HasDepth(mainDeviceHdl,16,gdDevType,1)))
  {
    DisableMenuItem(GetMenuRef(mDemonstration),0);
    return false;
  }
  else
  {
    gStartupPixelDepth = (**((**mainDeviceHdl).gdPMap)).pixelSize;
    return true;
  }
```

```
}

// ****************************************************************** doSetMonitorPixelDepth

void  doSetMonitorPixelDepth(SInt16 requiredDepth)
{
  GDHandle mainDeviceHdl;
  Str255   alertString;
  SInt16   currentPixelDepth;

  mainDeviceHdl = GetMainDevice();
  currentPixelDepth = (**((**mainDeviceHdl).gdPMap)).pixelSize;

  if(currentPixelDepth != requiredDepth)
  {
    SetDepth(mainDeviceHdl,requiredDepth,gdDevType,1);
  }
  else
  {
    GetIndString(alertString,rIndexedStrings,sMonitorAtThatDepth);
    doMonitorAlert(alertString);
  }
}

// ************************************************************** doRestoreMonitorPixelDepth

void  doRestoreMonitorPixelDepth(void)
{
  GDHandle mainDeviceHdl;
  Str255   alertString;
  SInt16   pixelDepth;

  mainDeviceHdl = GetMainDevice();
  pixelDepth = (**((**mainDeviceHdl).gdPMap)).pixelSize;

  if(pixelDepth != gStartupPixelDepth)
  {
    GetIndString(alertString,rIndexedStrings,sRestoringMonitor);
    doMonitorAlert(alertString);
    SetDepth(mainDeviceHdl,gStartupPixelDepth,gdDevType,1);
  }
  else
  {
    GetIndString(alertString,rIndexedStrings,sMonitorAtStartDepth);
    doMonitorAlert(alertString);
  }
}

// ************************************************************************** doMonitorAlert

void  doMonitorAlert(Str255 labelText)
{
  SInt16 itemHit;

  StandardAlert(kAlertNoteAlert,labelText,NULL,NULL,&itemHit);
}

// *****************************************************************************************
```

Demonstration Program PreQuickDraw Comments

When this program is run, the user should:

- Drag the window to various positions on the main screen, noting, on Mac OS 8/9 only, the changes to the coordinates of the pixel map's boundary rectangle. (On Mac OS X these coordinates represent the bounds of the Core Graphics window that backs the Carbon window, not the screen.)

- Change between the available monitor resolutions, noting the changes in the bytes per row and total pixel image bytes figures displayed in the window.

- Using the Demonstration menu, change between the available pixel depths, noting the changes to the pixel depth and total pixel image bytes figures, and the background colour values, displayed in the window.

- Note that, when a pixel depth of 8 is set on a direct device, the device creates a CLUT and operates like a direct device. In this case, the background colour value is the colour table entry (index), and the relevant colour in the colour table display is framed in white.

On Mac OS 8/9, if the user's monitor is set to thousands or millions of colours when the program is run for the first time, the colour table will not be built. It will be built when the user first sets the pixel depth to 8 (256 colours).

main

The call to doCheckMonitor determines whether the monitor can support a pixel depth of at least 16. If it cannot, the Demonstration menu is disabled, false is returned, and an alert is displayed advising the user that the Demonstration menu will be unavailable. If the monitor can support a pixel depth of at least 16, the current pixel depth is assigned to the global variable gStartupPixelDepth.

doEvents

In the case of a mouse-down event, in the inDrag case, when the user releases the mouse button, the window is invalidated, causing it to be redrawn.

doDisplayInformation

At the first two lines, RGB colours are assigned to the window's graphics port's rgbFgColor and rgbBkColor fields. The call to EraseRect causes the content region to be filled with the background colour.

Get Device List

The call to GetDeviceList gets a handle to the first GDevice structure in the device list. The device list is then "walked" in the while loop. For every video device found in the list, the variable videoDeviceCount is incremented. GetNextDevice gets a handle to the next device in the device list.

Get Main Device

GetMainDevice gets a handle to the startup device, that is, the device on which the menu bar appears.

Following the call to MoveTo, the gdDevType bit is tested to determine whether the main (startup) device is a colour or black-and-white device.

In the next block, the gdType field of the GDevice structure is examined to determine whether the device is an indexed device with a variable CLUT, an indexed device with a fixed CLUT, or a direct device (or a direct device set to display 256 colours or less and, as a consequence, acting like an indexed device).

Get Handle to Pixel Map

The call to GetWindowPort gets the reference to the window's graphics port required by the call to GetPortPixMap. GetPortPixMap gets a handle to the pixel map. (The following line shows an alternative method of obtaining a handle to a pixel map, in this case from the GDevice structure.)

In the next block, GetPixDepth is called to get the pixel depth. (The following line shows an alternative method of obtaining the pixel depth, in this case from the GDevice structure.)

At the next block, the number of bytes in each row in the pixel map is determined. (The high bit in the rowBytes field of the PixMap structure is a flag which indicates whether the data structure is a PixMap structure or a BitMap structure.)

Get Device's Boundary Rectangle

At the first line of this block, the device's boundary rectangle is extracted from the GDevice structure's gdRect field.

At the next block, the bytes per row value is multiplied by the height of the boundary rectangle to arrive at the total number of bytes in the pixel image.

The boundary rectangle's top, left, bottom, and right coordinates are then drawn in the window.

Get and Display Pixel Map's Boundary Rectangle

The call to GetPixBounds gets the pixel map's bounding rectangle. The rectangle's top, left, bottom, and right coordinates are then drawn in the window.

Get and Display RGB Components of Requested Background Colour

The second line of this block calls GetBackColor to get the graphics port's background colour. The red, green, and blue values are then printed in the window.

If Direct Device, Get and Display RGB Components of Colour Returned by GetCPixel

If the device is a direct device, GetCPixel is called to get the colour of a pixel in the window drawn with the background colour. The red, green and blue values are then printed in the window.

If the device is not a direct device, some preparatory text is drawn in the window.

Get Handle To Colour Table

The first and fourth lines get a handle to the colour table in the GDevice structure's pixel map and the number of entries in that table. (Note that the ctSize field of the ColorTable structure contains the number of table entries minus one.)

On Mac OS 8/9, QuickDraw only calls the Color Manager to build the colour table if the device is an indexed device (or a direct device acting as an indexed device). Thus, on Mac OS 8/9, there will only be a dummy entry in the colour table unless the monitor is an indexed device or a direct divice set to display 256 colours or less.

The final block paints small coloured rectangles for each entry in the colour table. If the main device is an indexed device (or if it is a direct device set to display 256 colours or less), the colour table entry being used as the best match for the requested background colour is outlined in white and the index value is drawn.

doCheckMonitor

doCheckMonitor is called at program start to determine whether the main device supports at least 16-bit colour and, if it does, to assign the main device's pixel depth at startup to the global variable gStartupPixelDepth.

The call to GetMainDevice gets a handle to the main device's GDevice structure. The function HasDepth is used to determine whether the device supports at least 16-bit colour. If it does not, the Demonstration menu is disabled and false is returned. If it does, the pixel depth is extracted from the pixelSize field of the PixMap structure in the GDevice structure and assigned to the global variable gStartupPixelDepth.

doSetMonitorPixelDepth

doSetMonitorPixelDepth is called when one of the the first three items in the Demonstration menu is chosen.

If the current pixel depth determined at the first two lines is not equal to the required new depth, SetDepth is called to set the main device's pixel depth to the required depth.
If the current pixel depth is equal to the required pixel depth, an alert is displayed advising the user that the device is currently set to that pixel depth.

doRestoreMonitorPixelDepth

doRestoreMonitorPixelDepth is called, when the last item in the Demonstration menu is chosen, to reset the main device's pixel depth to the startup pixel depth.

If the current pixel depth determined at the first two lines is not equal to the startup pixel depth, a string is retrieved from a 'STR#' resource and passed to the function doMonitorAlert, which displays a movable modal alert box advising the user that the monitor's bit depth is about to be changed to the startup pixel depth. When the user dismisses the alert box, SetDepth sets the main device's pixel depth to the startup pixel depth.

If the current pixel depth is the startup pixel depth, the last two lines display an alert box advising the user that the device is currently set to that pixel depth.