11

Version 1.2 (Frozen)

COLOR QUICKDRAW

Includes Demonstration Program ColorQuickDrawPascal

Introduction

Color QuickDraw is a collection of system software routines your application can use to display hundreds, thousands and even millions of colours on screens with those capabilities. Only those older Macintoshes based on the Motorola 68000 processor provide no support for Color QuickDraw.

You can draw into a colour graphics port using the eight predefined colours provided by basic QuickDraw. Color QuickDraw, however, provides for a greatly increased number of colours, the actual number available to your application depending on the user's computer system. In addition, Color QuickDraw allows you to define your own colours, and it provides a consistent way for your application to deal with colour regardless of the user's screen and software configuration.

RGB Colours

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

You specify a colour 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 colours. The `RGBColor` data type is defined as follows:

```
TYPE
  RGBColor = record
  red:       integer;     {magnitude of red component}
  green:     integer;     {magnitude of green component}
  blue:      integer;     {magnitude of blue component}
  end;
```

The Colour Drawing Environment - Colour Graphics Ports

A colour graphics port is automatically created when you use the Window Manager functions `GetNewCWindow` and `NewCWindow`. Colour graphics ports are also automatically created when your application provides the colour-awareness resources `'dctb'` and `'actb'` and then uses the Dialog Manager routines `GetNewDialog` and `Alert`.

A colour graphics port is defined in a `CGrafPort` record:

```
type
CGrafPort = record
  device:      integer;       {Device-specific information.}
  portPixMap:  PixMapHandle;  {Handle to pixel map.}
  portVersion: integer;       {Flags.}
  grafVars:    Handle;        {Handle to additional colour fields.}
  chExtra:     integer;       {Fractional horizontal pen position.}
  portRect:    Rect;          {Port Rectangle.}
  visRgn:      RgnHandle;     {Visible region.}
  clipRgn:     RgnHandle;     {Clipping region.}
  bkPixPat:    PixPatHandle;  {Background pattern.}
  rgbFgColor:  RGBColor;      {RGB components of fg.}
  rgbBkColor:  RGBColor;      {RGB components of bk.}
  pnLoc:       Point;         {Pen location.}
  pnSize:      Point;         {Pen size.}
  pnMode:      integer;       {Pen 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;       {Transfer mode for text.}
  txSize:      integer;       {Font size for text.}
  spExtra:     Fixed;         {Extra width added to space characters.}
  fgColor:     longint;       {Foreground colour.}
  bkColor:     longint;       {Background colour.}
  colrBit:     integer;       {Colour bit (reserved).}
  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;   {Pointer to low-level drawing routines.}
  end;

CGrafPtr = ^CGrafPort;
CWindowPtr = CGrafPtr;
```

Differences Between a `CGrafPort` Record and a `GrafPort` Record

A `CGrafPort` record is the same size as a `GrafPort` record. The important differences between these two data types are as follows:

- 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 four-byte `portPixMap` field, which contains a handle to a `PixMap` record (see Fig 1).

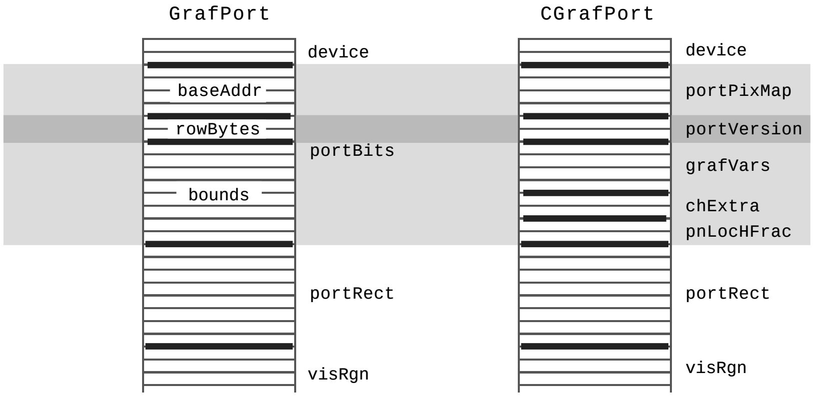

FIG 1 - FIRST 27 BYTES OF GrafPort AND CGrafPort RECORDS

- In what would be the `rowBytes` field of the `BitMap` record in the `portBits` field of the `GrafPort` record, a `CGrafPort` record has a two-byte `portVersion` field (see Fig 1) in which the two high bits are always set. QuickDraw uses these two bits to distinguish `CGrafPort` records from `GrafPort` records. (In `GrafPort` records, the two 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 (see Fig 1). The `GrafVars` records contains colour information used by Color QuickDraw and the Palette Manager.

- Following the `grafVars` field are the `chExtra` field, which holds the width of non-space characters in a font, and the `pnLocHFrac` field, which holds the fractional horizontal pen position used when drawing text.

- In a `GrafPort` record, `the bkPat`, `fillPat`, and `pnPat` fields hold eight-byte bit patterns. In a `CGrafPort` record, these fields are partly replaced by three four-byte handles to pixel patterns. The resulting 12 bytes of additional space are taken up by the `rgbFgColor` and `rgbBkColor` fields, which contain six-byte `RGBColor` records specifying the optimal foreground and background colours for the colour graphics port. (See Fig 2.) Note that the closest matching available colours, which Color QuickDraw actually uses for the foreground and background, are stored in the `fgColor` and `bkColor` fields of the `CGrafPort` record.

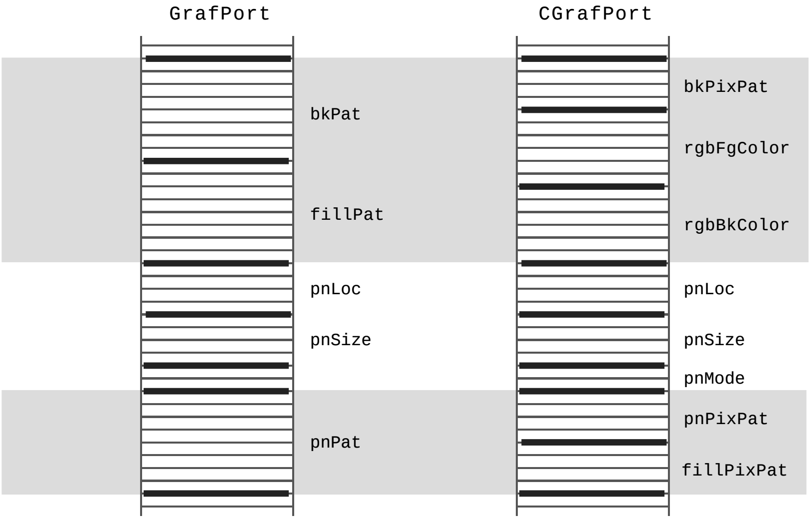

FIG 2 - BYTES 27 - 62 OF GrafPort AND CGrafPort RECORDS

Working with a `CGrafPort` record is much like working with 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 what type it is.

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

```
CGrafPtr^.myPort;
SetPort(GrafPtr(myPort));
```

You can use all QuickDraw drawing commands to draw 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 do not take advantage of Color QuickDraw's colour features.

Pixel Maps

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

The representation of a colour image in memory is a **pixel image**, analogous to the bit image used by basic QuickDraw. A `PixMap` record contains a pointer to a pixel image, its dimensions, storage format, depth, resolution, and colour usage.

The `PixMap` record is as follows:

```
type
PixMap = record
  baseAddr:   Ptr;        {Pointer to image data.}
  rowBytes:   integer;    {Flags, and bytes in a row.}
  bounds:     Rect;       {Boundary rectangle.}
  pmVersion:  integer;    {Pixel Map version number.}
  packType:   integer;    {Defines packing format.}
  packSize:   longint;    {Size of data in packed state.}
  hRes:       Fixed;      {Horizontal resolution in dots per inch.}
  vRes:       Fixed;      {Vertical resolution in dots per inch.}
  pixelType:  integer;    {Format of pixel image.}
  pixelSize:  integer;    {Physical bits per pixel.}
  cmpCount:   integer;    {Number of components in each pixel.}
  cmpSize:    integer;    {Number of bits in each component.}
  planeBytes: longint;    {Offset to next plane.}
  pmTable:    CTabHandle; {Handle to a colour table for this image.}
  pmReserved: longint;    {Reserved for future use. Must be 0.}
  end;

PixMapPtr = ^PixMap;
PixMapHandle = ^PixMapPtr;
```

Field Descriptions

`baseAddr`	Contains a pointer to the beginning of the onscreen pixel image for a pixel map. The pixel image that appears on the screen is normally stored on a graphics card rather than in main memory. (Note that there can be several pixel maps pointing to the same pixel image, each imposing its own coordinate system on it.)
`rowBytes`	The offset in bytes from one row of the image to the next. The value must be even and less than `$4000`. For best performance it should be a multiple of 4. The high two bits are used as flags. If bit 15 = 1, the data structure pointed to is a `PixMap` record, otherwise it is a `BitMap` record.
`bounds`	As with a bitmap, the pixel map's boundary rectangle is initially set to the size of the main screen.
`pmVersion`	The version number of Color QuickDraw that created this `PixMap` record. The value is normally 0. If it is 4, Color QuickDraw treats the `baseAddr` field as 32-bit clean. 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 (no packing) and values of 1 to 4 for packing direct pixels.
`packSize`	The size of the packed image in bytes. (When `packType` is 0, this field is set to 0.)
`hRes`	The horizontal resolution of the image in pixels per inch, abbreviated as dpi (dots per inch). The value of this field is of type `Fixed`. By default, the dpi is 72, but Color QuickDraw supports `PixMap` records of other resolutions. For example, `PixMap` records for scanners can have dpi resolutions of 150, 200, 300, or greater.
`vRes`	Describes the vertical resolution. (See `hRes`).
`pixelType`	Specifies the format (indexed or direct) used to hold the pixels in the image. For indexed devices, the value is 0. For direct devices, the value is 16, which can be represented by the constant `RGBDirect`.
`pixelSize`	Specifies the **pixel depth**, that is, the number of bits per pixel in the pixel image. 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.) 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.)[1]

`cmpCount`	Together with `cmpSize`, describes how the pixel values are organised. For pixels on indexed devices, the colour component count is 1 (for the index into the graphic's device's CLUT, where the colours are stored). For pixels in direct devices, the colour component count is 3 (for the red, green and blue components of each pixel).
`cmpSize`	Specifies how large each colour component is. For indexed devices, it is the same value as that in the `pixelSize` field, that is, 1, 2, 4, or 8 bits. 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).
`planeBytes`	Specifies an offset in bytes from one plane to another. Since Color QuickDraw does not support multiple-plane images, the value of this field is always 0.
`pmTable`	Contains a handle to the `ColorTable` record. `ColorTable` records define the colours available for pixel images on indexed devices. (The Color Manager stores a colour table for the currently available colours in the graphic's device's CLUT. You use the Palette Manager to assign different colour tables to your different windows.) You can create colour tables using either `ColorTable` records or `'clut'` resources. Pixel images on direct devices do not need a colour table because the colours are stored right in the pixel values. In such cases, `pmTable` points to a dummy colour table.

Translation of RGB Colours to Pixel Values

The `baseAddr` field of the `CGrafPort` record 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, Color QuickDraw translates that colour 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 colour. The translation from the colour you specify in an `RGBColor` record 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, Color 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, Color QuickDraw truncates the least significant bits from the red, green and blue fields of the `RGBColor` record. The result becomes the pixel value that Color QuickDraw sends to the graphics device.

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

---

[1] Note that, when a user uses the Monitors control panel to set a 16-bit or 32-bit device to use 2, 4, 16 or 256 colours as a grayscale or colour device, the direct device creates a CLUT and operates like an indexed device.

Derivation of Pixel Values on Indexed Devices

Fig 3 shows the translation of an `RGBColor` record to an 8-bit pixel value on an indexed device.

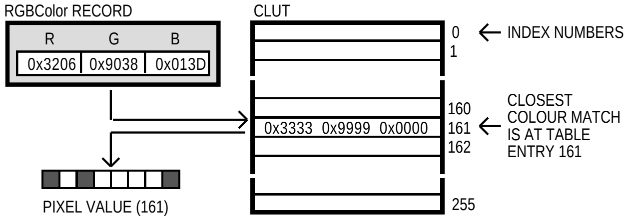

FIG 3 - TRANSLATING AN RGBColor RECORD TO AN 8-BIT PIXEL VALUE ON AN INDEXED DEVICE

The application might later use `GetCPixel` to determine the colour of a particular pixel. As shown at Fig 4, the Color Manager uses the index number stored as the pixel value to find the `RGBColor` record stored in the CLUT for that pixel's colour. Also as shown at Fig 4, this is not necessarily the exact colour first specified.

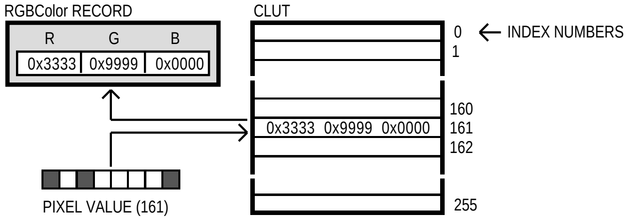

FIG 4 - TRANSLATING AN 8-BIT PIXEL VALUE ON AN IDEXED DEVICE TO AN RGBColor RECORD

Derivation of Pixel Values on Direct Devices

Fig 5 shows how Color QuickDraw converts an `RBGColor` record into a 16-bit pixel value on a direct device by storing the most significant 5 bits of each 16-bit field of the 48-bit `RGBColor` record in the lower 15 bits of the pixel value, leaving an unused high bit. Fig 5 also 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, from the original 48-bit value.

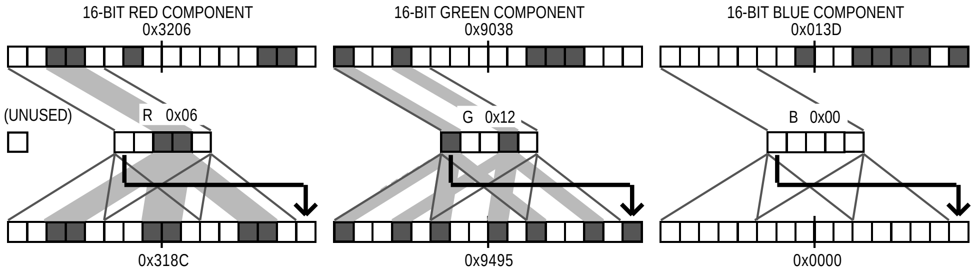

FIG 5 - TRANSLATING AN RGBColor RECORD TO A 16 BIT PIXEL VALUE,
AND FROM A 16-BIT PIXEL VALUE TO AN RGBRecord, ON A DIRECT DEVICE

Fig 6 shows how Color QuickDraw converts an `RBGColor` record into a 32-bit pixel value on a direct device by storing the most significant 8 bits of each 16-bit field of the record into the lower 3 bytes of the pixel value, leaving 8 unused bits in the high byte of the pixel value. Fig 6 also shows how Color QuickDraw expands a 32-bit pixel value to an `RBGColor` 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 `RBGColor` record.

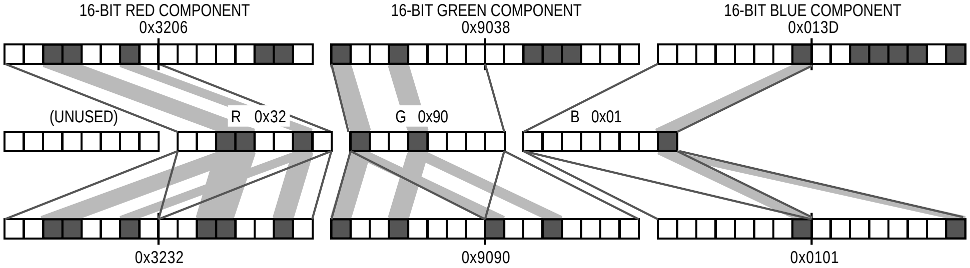

FIG 6 - TRANSLATING AN RGBColor RECORD TO A 32 BIT PIXEL VALUE, AND FROM A 32-BIT PIXEL VALUE TO AN RGBRecord, ON A DIRECT DEVICE

Colours on Grayscale Screens

When Color 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 colour graphics device that the user sets to grayscale by using the Monitors control panel. For such a graphics device, Colour QuickDraw places an evenly spaced set of grays in the graphics device's CLUT.

By using the `GetCTable` function, your application can obtain the default colour tables for various graphics devices, including grayscale devices.

Pixel Patterns

Color QuickDraw supplements the black-and-white bit patterns of basic QuickDraw with **pixel patterns**. 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 simpler and more convenient to store them in resources of type `'ppat'`.

Pen Pixel Pattern

As with bit patterns, your application can use pixel patterns to draw lines and shapes on the screen. In a colour graphics port, the graphics pen has a pixel pattern specified in the `pnPixPat` field of the `CGrafPort` record. The pixels in the pattern interact with the pixels in the pixel map according to the pattern mode of the graphics pen.

Initially, every graphics pen is assigned an all black pattern, but you can use `PenPixPat` to assign a different pixel pattern to the graphics pen.

`FrameRect`, `FrameRoundRect`, `FrameArc`, `FramePoly`, `FrameRgn`, `PaintRect`, `PaintRoundRect`, `PaintArc`, `PaintPoly`, and `PaintRgn` are used to draw with the pattern specified in the `pnPixPat` field.

Fill Pixel Pattern

`FillCRect`, `FillCRoundRect`, `FillCArc`, `FillCPoly`, and `FillCRgn` are used to draw shapes with the pixel pattern specified as the parameter in the call to these routines. The pixel pattern specified in the call is stored in the `fillPixPat` field of the `CGrafPort` record.

Background Pixel Pattern

The colour graphics port also has a background pattern which is used when an area is erased (for example, by `EraseRect`, `EraseRoundRect`, `EraseArc`, `ErasePoly`, and `EraseRgn`) and when pixels are scrolled out of an area by `ScrollRect`. The background pattern is stored in the `bkPixPat` field of the `CGrafPort` record. It can be changed using `BackPixPat`.

Creating Pseudo Colours With Pixel Patterns

Pixel patterns can be used to create colours otherwise unavailable on indexed devices. For example, if your application draws to an indexed device that supports 4 bits per pixel, your application has a maximum of 16 colours available. However, if your application uses `MakeRGBPat` to create patterns that use these 16 colours in different combinations, and then draws using that pattern, your application can have as many as 109 additional (pseudo) colours at its disposal.

Testing For the Existence of Color QuickDraw

Before using Color QuickDraw routines, your application should check for the existence of Color QuickDraw by using the `Gestalt` function. The `Gestalt` function is used to acquire information about the operating environment[2]. It has two parameters: a **selector** and a **response**.

When testing for the existence of Color QuickDraw, `Gestalt` should be called with the `gestaltQuickDrawVersion` selector. The low-order word in the four-byte value returned in the `response` parameter contains QuickDraw version data. In that low-order word, the high byte gives the major revision number and the low byte gives the minor revision number. If the value returned in the `response` parameter is equal to the constant `gestalt32BitQD13`, then the system supports the System 7 version of Color QuickDraw.

The following are the constants, and the values they represent, which indicate the various versions of Color QuickDraw:

Constant	Value	Version
`gestalt8BitQD`	`$100`	8-bit Color QuickDraw
`gestalt32BitQD`	`$200`	32-bit Color QuickDraw
`gestalt32BitQD11`	`$210`	32-bit Color QuickDraw v1.1
`gestalt32BitQD12`	`$220`	32-bit Color QuickDraw v1.2
`gestalt32BitQD13`	`$230`	System 7: 32-bit Color QuickDraw v1.3

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

Constant	Value	Feature
`gestaltHasColor`	`0`	Color QuickDraw is present.
`gestaltHasDeepGWorlds`	`1`	GWorlds deeper than one bit.
`gestaltHasDirectPixMaps`	`2`	PixMaps can be direct - 16-bit or 32-bit
`gestaltHasGrayishTextOr`	`3`	Supports text mode `grayishTextOr`

Working with Color QuickDraw

All of the basic QuickDraw routines work with Color QuickDraw.

Creating Colour Graphics Ports

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

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

---

[2] The `Gestalt` function is explained in detail at Chapter 22— Miscellany.

type `CWindowPtr`. A pointer of type `WindowPtr` points to a `GrafPort` record. Thus, if you want to check the fields of the colour graphics port associated with a window, you must coerce the pointer to the `GrafPort` record into a pointer to a `CGrafPort` record.

Drawing with Different Foreground Colours

If your application uses the Palette Manager, it should set the foreground and background colours with the Palette Manager routines `PmForeColor` and `PmBackColor`. Otherwise, you application should use Color QuickDraw's `RGBForeColor` and `RGBBackColor` routines.

To specify a foreground colour, create an `RGBColor` record and use that record as the `RGBForeColor` parameter in the call, for example:

```
darkBlue : RGBColor;
...
darkBlue.red   := $0000;
darkBlue.green := $0000;
darkBlue.blue  := $9999;

RGBForeColor(darkBlue);
```

`RGBForeColor` supplies the `rgbFgColor` field of the `CGrafPort` record with this record, and it places the closest available match in the `fgColor` field. The colour in the `fgColor` field is the colour actually used as the foreground colour.

`RGBForeColor` and `RGBBackColor` also work in basic graphics ports created in System 7.

Drawing and Filling with Pixel Patterns

If you wish to draw with a colour other than the foreground colour, you can give the graphics pen a pixel pattern using `PenPixPat`. To fill shapes with pixel patterns, you can use `FillCRect`, `FillCRoundRect`, `FillCOval`, `FillCArc`, `FillCPoly`, and `FillCRgn`.[3]

You define a pixel pattern in a `'ppat'` resource. To retrieve the pixel pattern stored in the `'ppat'` resource, you use the `GetPixPat` function. The handle to a `pixPat` data structure returned by `GetPixPat` may then be used in a call to `PenPixPat` to assign the pattern to the pen.

The following is an example of the use of pixel patterns for painting and filling:

```
theRect : Rect ;
penPattern,fillPattern : PixPatHandle;
...
penPattern := GetPixPat(128);
PenPixPat(penPattern);
SetRect(theRect, 20, 20, 70, 70);
PaintRect(theRect);
DisposePixPat(penPattern);

fillPattern := GetPixPat(129);
SetRect(theRect, 90, 20, 140, 70);
FillCRect(theRect, fillPattern);
DisposePixPat(fillPattern);
```

Using Bit Patterns in Colour Graphics Ports

When you use basic QuickDraw's `PenPat` and `BackPat` routines in a colour graphics port, Color QuickDraw constructs a pixel pattern equivalent to the bit pattern you specify to `PenPat` and `BackPat`. The resulting pen pattern and background pattern use the graphics port's current foreground and background colours.

---

[3] Note that, because a pixel pattern already contains colour, Color QuickDraw ignores the foreground and background colours when your application draws with a pixel pattern.

Boolean Pattern Modes with Colour Pixels

Pattern modes apply to the drawing of lines and shapes. When you use pattern modes in pixel maps with depths greater than 1 bit, Color QuickDraw uses the foreground and background colour when transferring bit patterns. For example, the `patCopy` mode applies the foreground colour to every destination pixel that corresponds to a black pixel in a bit pattern, and it applies the background colour to every destination pixel that corresponds to a white pixel in a bit pattern.

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

Copying Pixels Between Colour Graphics Ports

Color QuickDraw provides extra capabilities for the `CopyBits`, `CopyMask`, and `CopyDeepMask` image-processing routines described at Chapter 10 — Basic QuickDraw. In basic QuickDraw, `CopyBits`, `CopyMask`, and `CopyDeepMask` are used to copy bit images between two basic graphics ports. In Color QuickDraw, you can also use these routines to copy pixel images between two colour graphics ports. In addition, the masks used by `CopyMask` and `CopyDeepMask` may be another pixel map whose pixels indicate proportionate weights of the colours for the source and destination pixels.

Distinguishing Between Bit Maps and Pixel Maps

`CopyBits`, `CopyMask`, and `CopyDeepMask` expect a pointer to a bitmap in their source and destination parameters. Accordingly, when you use these routines to copy pixel images between colour 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. For example, if your application copies a pixel image from a colour graphics port called, say, `myColourPort`, you could specify `GrafPtr(myColourPort)^.portBits` in the `srcBits` parameter.

All this works because:

- In a `CGrafPort` record, the two high bits of the `portVersion` field are always set.
- These bits in a `GrafPort` record are the two high bits in `portBits.rowBytes` field, which are always clear.
- By looking at these bits, `CopyBits`, `CopyMask`, and `CopyDeepMask` can establish that you have passed the routines a handle to a pixel map rather than the base address of a bitmap.

CopyMask

With `CopyMask`, you supply a pixel map to act as the copying mask. The values of pixels in the mask act as weights that proportionally select between source and destination pixel values.

On indexed devices, pixel images are always copied using the colour table of the source `PixMap` record for source colour information, and using the colour table of the current `GDevice` record for destination colour information. The colour table attached to the destination `PixMap` is ignored.

When the `PixMap` record for the mask is 1 bit deep, it has the same effect as a bitmap mask, that is, a black bit in the mask means that the destination pixel will take the colour of the source pixel and a white bit in the mask means that the destination pixel is to retain its current colour. When masks have `PixMap` records with pixel depths greater than 1, Colour QuickDraw takes a weighted average between the colours in the source and destination `PixMap` records. Within each pixel, the calculation is done in RGB colour, on a colour component basis. As an example, 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 Source Modes with Colour Pixels

When you use `CopyBits`, `CopyMask`, and `CopyDeepMask` to transfer images between pixel maps with depths greater than 1 bit, Color QuickDraw performs the Boolean transfer operations as follows:

Source Mode	Action On Destination Pixel		
	If source pixel is black	**If source pixel is white**	**If source pixel is any other colour**
`srcCopy`	Apply foreground colour	Apply background colour	Apply weighted portions of foreground and background colours
`notSrcCopy`	Apply background colour	Apply foreground colour	Apply weighted portions of foreground and background colours
`srcOr`	Apply foreground colour	Leave alone	Apply weighted portions of foreground colour
`notSrcOr`	Leave alone	Apply foreground colour	Apply weighted portions of foreground colour
`srcXor`	Invert (undefined for coloured destination pixel)	Leave alone	Leave alone
`notSrcXor`	Leave alone	Invert (undefined for coloured destination pixel)	Leave alone
`srcBic`	Apply background colour	Leave alone	Apply weighted portion background colour
`notSrcBic`	Leave alone	Apply background colour	Apply weighted portion background colour

In general, with pixel images, you will probably want to use `srcCopy` mode or one of the arithmetic transfer modes (see below).

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

- The `notSrcCopy` mode reverses the foreground and background colours.
- 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 colour is black when you use `srcOr` and `notSrcOr` modes.
- If the background colour is white when you use the `srcBic` mode, the black portions of the source are erased, resulting in white in the destination pixel map.

Applying a foreground colour other than black or a background colour other than white to the pixel can produce an unexpected result. For consistent results, set the foreground colour to black and the background colour to white before using `CopyBits`, `CopyMask`, or `CopyDeepMask`. (That said, using `RGBForeColor` and `RGBBackColor` to set foreground and background colours to something other than black or white can achieve some interesting colouration effects.)e e

Dithering

You can use **dithering** with `CopyBits` and `CopyDeepMask`. Dithering is a technique used by these routines to mix existing colours together to create the illusion of a third colour that may be unavailable on an indexed device, and to improve images that you shrink when copying them from a direct device to an indexed device.

You can add dithering to any source mode by adding the following constant, or the value it represents, to the source mode:

```
ditherCopy := 64   {Add to source mode for dithering.}
```

If you specify a destination rectangle that is smaller than the source rectangle when using `CopyBits`, `CopyMask`, `CopyDeepMask` on an 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 when you explicitly specify dithering.

Dithering has drawbacks. Firstly, it slows the drawing operation. Secondly, a clipped dithering operation does not provide pixel-for-pixel equivalence to the same unclipped dithering operation.

Arithmetic Transfer Modes

In addition to the Boolean transfer modes, 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 colours rather than with colour table indexes. The arithmetic transfer modes are as follows:

Constant	**Value**	**Description**
`blend`	32	Replace destination pixel with a blend of the source and destination pixel colours. If the destination is a bitmap or 1-bit pixel image, revert to `srcCopy` mode.
`addPin`	33	Replace destination pixel with the sum of the source and destination pixel colours up to a maximum allowable value. If the destination is a bitmap or 1-bit pixel image, revert to `srcBic` mode.
`addOver`	34	Replace destination pixel with the sum of the source and destination pixel colours, 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 image, revert to `srcXor` mode.
`subPin`	35	Replace destination pixel with the difference of the source and destination pixel colours, but not less than a minimum allowable value. If the destination is a bitmap or 1-bit pixel image, revert to `srcOr` mode.
`transparent`	36	Replace the source and destination pixel with the source pixel if the source pixel is not equal to the background colour.
`addMax`	37	Compare the source and destination pixels, and replace the destination pixel with the colour containing the greater saturation of each of the RGB components. If the destination is a bitmap or 1-bit pixel image, revert to `srcBic` mode.
`subOver`	38	Replace destination pixel with the difference of the source and destination pixel colours, but if the value of the red, green or blue is less than 0, add the negative result to 65,536. If the destination is a bitmap or 1-bit pixel image, revert to `srcXor` mode.
`adMin`	39	Compare the source and destination pixels, and replace the destination pixel with the colour containing the lesser saturation of each of the RGB components. If the destination is a bitmap or 1-bit pixel image, revert to `srcOr` mode

You can use the arithmetic modes for both drawing and image transfer operations, that is, your application can pass them in parameters to `PenMode` and `TextMode` as well as `CopyBits` and `CopyDeepMask`.

Highlighting

When using basic QuickDraw, you can use `InvertRect`, or any other image-copying routine that uses the `srcXor` source mode, to **invert** objects on the screen. Inverting simply reverses the colours of all pixels within the specified rectangle. Although this procedure can also be used on colour pixels in colour graphics ports, the results are predictable only with direct pixels or 1-bit pixel maps. Accordingly, with Color QuickDraw, you should use **highlighting**, rather than inverting, when selecting and deselecting objects such as text or graphics.

TextEdit, for example, uses highlighting to indicate selected text. If the highlight colour is blue, TextEdit draws the selected text, then uses `InvertRgn` to produce a blue background for the text.

The **system highlight colour**, which can be changed by the user using the Colour control panel, is stored in a low memory global represented by the symbolic name `HiliteRGB`. It can be retrieved using `LMGetHiliteRGB`. Basic graphics ports use this colour as the highlight colour. In the case of a colour graphics port, you can override the default colour using `HiliteColor`. (Note that the current colour is copied to the `rgbHiliteColor` field of the `GrafVars` record, a handle to which is stored in the `grafVars` field of the `CGrafPort` record.)

Color QuickDraw implements highlighting by replacing the background colour with the highlight colour. Another low memory global, represented by the symbolic name `HiliteMode`, contains a byte which represents the current highlight mode. One bit in that byte, represented by the constant `pHiliteBit`, is used to toggle the background and highlight colours.

Color QuickDraw resets the highlight bit after performing each drawing operation, so your application should always clear the highlight bit immediately before calling `InvertRgn` (or any of the other drawing or image-copying routine that uses the `patXor` or `srcXor` transfer mode.) The highlight mode can be retrieved and set using `LMGetHiliteMode` and `LMSetHiliteMode`, and `BitClr` may be used to clear the highlight bit:

```
hiliteMode : UInt8;
...
hiliteMode := LMGetHiliteMode;
BitClr(hiliteMode, pHiliteBit);
LMSetHiliteMode(hiliteMode);
```

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:

```
hilite := 50  {Add to source or pattern mode for highlighting.}
```

Color QuickDraw and Text

When drawing text using Color QuickDraw, the following information, in addition to that in Chapter 10 — Basic QuickDraw, is relevant:

- As previously stated, there is an additional text-related field in the colour graphics port record (the `chExtra` field. The value in this field may be changed using `CharExtra`.
- The arithmetic transfer modes apply to the drawing of text as well as other forms of graphics.
- When the default transfer mode (`srcOr`) is used, the colour of the glyph is determined by the foreground colour.
- The non-standard text drawing transfer mode `grayishTextOr` (which is useful for displaying disabled user interface items) produces a blend of the foreground and background colours on a colour destination device.

Main Color QuickDraw Constants, Data Types and Routines

Constants

Checking for Color QuickDraw and its Features

```
gestalt8BitQD              = $100    8-bit Color QuickDraw.
gestalt32BitQD             = $200    32-bit Color QuickDraw.
gestalt32BitQD11           = $210    32-bit Color QuickDraw v1.1.
gestalt32BitQD12           = $220    32-bit Color QuickDraw v1.2.
gestalt32BitQD13           = $230    System 7: 32-bit Color QuickDraw v1.3.
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 bits.
gestaltHasGrayishTextOr    = 3       Supports text mode grayishTextOr.
```

Arithmetic Transfer Modes

```
blend        = 32
addPin       = 33
addOver      = 34
subPin       = 35
transparent  = 36
```

```
addMax       = 37
subOver      = 38
adMin        = 39
ditherCopy   = 64
```

Highlighting

```
hilite       = 50
hiliteBit    = 7
pHiliteBit   = 0
```

Resource ID of `'clut'` Resource for Default QuickDraw Colours

```
defQDColors  = 127
```

Pixel Type

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

Data Types

```
PixelType = SInt8;
```

CGrafPort

```
CGrafPort = record
  device:         integer;      {Device-specific information.}
  portPixMap:     PixMapHandle; {Handle to pixel map.}
  portVersion:    integer;      {Flags.}
  grafVars:       Handle;       {Handle to additional colour fields.}
  chExtra:        integer;      {Extra width added to non-space characters.}
  pnLocHFrac:     integer;      {Fractional horizontal pen position.}
  portRect:       Rect;         {Port Rectangle.}
  visRgn:         RgnHandle;    {Visible region.}
  clipRgn:        RgnHandle;    {Clipping region.}
  bkPixPat:       PixPatHandle; {Background pattern.}
  rgbFgColor:     RGBColor;     {RGB components of fg.}
  rgbBkColor:     RGBColor;     {RGB components of bk.}
  pnLoc:          Point;        {Pen location.}
  pnSize:         Point;        {Pen size.}
  pnMode:         integer;      {Pen 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;      {Transfer mode for text.}
  txSize:         integer;      {Font size for text.}
  spExtra:        Fixed;        {Extra width added to space charcaters.}
  fgColor:        longint;      {Foreground colour.}
  bkColor:        longint;      {Background colour.}
  colrBit:        integer;      {Colour bit (reserved).}
  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;  {Pointer to low-level drawing routines.}
  end;

CGrafPtr = ^CGrafPort;
CWindowPtr = CGrafPtr;
```

PixMap

```
PixMap = record
  baseAddr:       Ptr;          {Pointer to image data.}
  rowBytes:       integer;      {Flags, and bytes in a row.}
  bounds:         Rect;         {Boundary rectangle.}
  pmVersion:      integer;      {Pixel Map version number.}
  packType:       integer;      {Defines packing format.}
  packSize:       longint;      {Size of data in packed state.}
  hRes:           Fixed;        {Horizontal resolution in dots per inch.}
  vRes:           Fixed;        {Vertical resolution in dots per inch.}
  pixelType:      integer;      {Format of pixel image.}
```

```
  pixelSize:      integer;      {Physical bits per pixel.}
  cmpCount:       integer;      {Number of components in each pixel.}
  cmpSize:        integer;      {Number of bits in each component.}
  planeBytes:     longint;      {Offset to next plane.}
  pmTable:        CTabHandle;   {Handle to a colour table for this image.}
  pmReserved:     longint;      {Reserved for future use. Must be 0.}
  end;

PixMapPtr = ^PixMap;
PixMapHandle = ^PixMapPtr;
```

GrafVars

```
GrafVars = record
  rgbOpColor:     RGBColor;     {Colour for addPin, subPin and average.}
  rgbHiliteColor: RGBColor;     {Colour for highlighting.}
  pmFgColor:      Handle;       {Palette Handle for foreground colour.}
  pmFgIndex:      integer;      {Index value for foreground.}
  pmBkColor:      Handle;       {Palette Handle for background colour.}
  pmBkIndex:      integer;      {Index value for background.}
  pmFlags:        integer;      {Flags for Palette Manager.}
  end;

GVarPtr = ^GrafVars;
GVarHandle = ^GVarPtr;
```

ColorSpec

```
TYPE
  ColorSpec = record
  value:          integer;      {Index or other value.}
  rgb:            RGBColor;     {True colour.}
  end;

ColorSpecPtr = ^ColorSpec;
CSpecArray = array [0..0] of ColorSpec;
```

ColorTable

```
ColorTable = record
  ctSeed:         longint;      {Unique identifier for table.}
  ctFlags:        integer;      {High bit: 0 = PixMap; 1 = device.}
  ctSize:         integer;      {Number of entries in CTTable.}
  ctTable:        CSpecArray;   {Array [0..0] of ColorSpec.}
  end;

CTabPtr = ^ColorTable;
CTabHandle = ^CTabPtr;
```

PixPat

```
PixPat = record
  patType:        integer;      {Type of pattern.}
  patMap:         PixMapHandle; {The pattern's pixMap.}
  patData:        Handle;       {Pixmap's data.}
  patXData:       Handle;       {Expanded Pattern data.}
  patXValid:      integer;      {Flags whether expanded Pattern valid.}
  patXMap:        Handle;       {Handle to expanded Pattern data.}
  pat1Data:       Pattern;      {Old-Style pattern/RGB colour.}
  end;

PixPatPtr = ^PixPat;
PixPatHandle = ^PixPatPtr;
```

RGBColor

```
RGBColor = record
  red:            integer;      {Magnitude of red component.}
  green:          integer;      {Magnitude of green component.}
  blue:           integer;      {Magnitude of blue component.}
  end;

RGBColorPtr = ^RGBColor;
RGBColorHdl = ^RGBColorPtr;
```

Routines

Opening and Closing Colour Graphics Ports

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

Managing a Colour Graphics Pen

```
procedure  PenPixPat(pp: PixPatHandle);
```

Changing the Background Pixel pattern

```
procedure  BackPixPat(pp: PixPatHandle);
```

Drawing with Color QuickDraw Colours

```
procedure  RGBForeColor(var color: RGBColor);
procedure  RGBBackColor(var color: RGBColor);
procedure  SetCPixel(h: integer; v: integer; var cPix: RGBColor);
procedure  FillCRect(var r: Rect; pp: PixPatHandle);
procedure  FillCOval(var r: Rect; pp: PixPatHandle);
procedure  FillCRoundRect(var r: Rect; ovalWidth: integer; ovalHeight: integer;
           pp: PixPatHandle);
procedure  FillCArc(var r: Rect; startAngle: integer; arcAngle: integer; pp: PixPatHandle);
procedure  FillCRgn(rgn: RgnHandle; pp: PixPatHandle);
procedure  FillCPoly(poly: PolyHandle; pp: PixPatHandle);
procedure  OpColor(var color: RGBColor);
procedure  HiliteColor(var color: RGBColor);
```

Determining Current Colours and Best Intermediate Colours

```
procedure  GetForeColor(var color: RGBColor);
procedure  GetBackColor(var color: RGBColor);
procedure  GetCPixel(h: integer; v: integer; var cPix: RGBColor);
function   GetGray(device: GDHandle; var backGround: RGBColor;
           var foreGround: RGBColor): boolean;
```

Creating, Setting and Disposing of Pixel Maps

```
function   NewPixMap: PixMapHandle;
procedure  CopyPixMap(srcPM: PixMapHandle; dstPM: PixMapHandle);
procedure  SetPortPix(pm: PixMapHandle);
procedure  DisposePixMap(pm: PixMapHandle);
```

Creating and Disposing of Pixel Patterns

```
function   GetPixPat(patID: integer): PixPatHandle;
function   NewPixPat: PixPatHandle;
procedure  CopyPixPat(srcPP: PixPatHandle; dstPP: PixPatHandle);
procedure  MakeRGBPat(pp: PixPatHandle; var myColor: RGBColor);
procedure  DisposePixPat(pp: PixPatHandle);
```

Creating and Disposing of Colour Tables

```
function   GetCTable(ctID: integer): CTabHandle;
procedure  DisposeCTable(cTable: CTabHandle);
```

Retrieving Color QuickDraw Result Codes

```
function   QDError: integer;
```

Getting and Setting the Highlight Colour and HighLight Mode (Defined in LowMem.h)

```
procedure  LMGetHiliteRGB(var hiliteRGBValue: RGBColor);
procedure  LMSetHiliteRGB(var hiliteRGBValue: RGBColor);
function   LMGetHiliteMode : ByteParameter;
procedure  LMSetHiliteMode( value: ByteParameter );
```

Demonstration Program

```
{ ######################################################################################

// ColorQuickDrawPascal.p
// ######################################################################################
//
// This program:
//
// •  Opens a window in which the results of various basic Color QuickDraw drawing
//    operations are displayed.
//
//    Individual drawing operations are selected from a pull-down menu titled
//    'Demonstration'.)
//
// •  Quits when the user selects Quit from the File menu or clicks the window's close
//    box.
//
// The program utilises the following resources:
//
// •  An 'MBAR' resource and associated 'MENU' resources (preload, non-purgeable).
//
// •  'WIND' resources (purgeable) (initially visible) for the main window, and for small
//    windows used for the CopyDeepMask and Transfer Modes demonstrations.
//
// •  An 'ALRT' resource and associated 'DITL' resource (purgeable).
//
// •  Three 'PICT' resources (purgeable).
//
// •  Two 'pltt' resources (purgeable).
//
// •  Two 'ppat' resources (purgeable);
//
// •  A 'STR#' resource (purgeable).
//
// ###################################################################################### }

program ColorQuickDrawPascal(input, output);

{ ………………………………………………………………………………………………………………………… include the following Universal Interfaces }

uses

  Windows, Fonts, Menus, TextEdit, Quickdraw, Dialogs, QuickdrawText, Processes, Types,
  Memory, Events, TextUtils, ToolUtils, OSUtils, Devices, Palettes, QDOffscreen,
  Resources, LowMem, GestaltEqu, Segload;

{ …………………………………………………………………………………………………………………………………………………… define the following constants }

const

mApple = 128;
mFile = 129;
  iQuit = 11;
mDemonstration = 131;
  iBitPattern = 1;
  iPixelPattern = 2;
  iCopyDeepMask = 3;
  iTransferModes = 4;
  iHighlighting = 5;
  iColorTable = 6;
rWindow = 128;
rImageWindow = 129;
rMenubar = 128;
rAlert = 128;
rIndexedStrings = 128;
rPaletteBaseID = 128;
rPixelPattern1 = 128;
rPixelPattern2 = 129;
rPicture = 128;
sColorQuickdraw = 1;
sSettingMonitor = 2;
sNeedMonitor = 3;
sRestoringMonitor = 4;

kMaxLong = $7FFFFFFF;
```

```

{ ……………………………………………………………………………………………………………………………………………………………………………………………… global variables }

var

gDone : boolean;
gWindowPtr : WindowPtr;
gWhiteColour : RGBColor;
gBlackColour : RGBColor;
gOchreColour : RGBColor;
gGreenColour : RGBColor;
theErr, ignored : OSErr;
response : longint;
alertString : string;
menubarHdl : Handle;
menuHdl : MenuHandle;
eventRec : EventRecord;
gotEvent : boolean;

{ ###################################################################### DoInitManagers }

procedure DoInitManagers;

  begin
  MaxApplZone;
  MoreMasters;

  InitGraf(@qd.thePort);
  InitFonts;
  InitWindows;
  InitMenus;
  TEInit;
  InitDialogs(nil);

  InitCursor;
  FlushEvents(everyEvent, 0);
  end;
    {of procedure DoInitManagers}

{ ######################################################################## DoBitPattern }

procedure DoBitPattern;

  var
  a : integer;
  paletteHdl : PaletteHandle;
  theRect : Rect;
  sysPattern : Pattern;
  theString : string;

  begin
  for a := 0 to 1 do
    begin
    paletteHdl := GetNewPalette(rPaletteBaseID + a);
    SetPalette(gWindowPtr, paletteHdl, true);

    PmBackColor(2);
    FillRect(gWindowPtr^.portRect, qd.white);

    SetRect(theRect, 10, 30, 245, 150);
    PenSize(10, 20);
    GetIndPattern(sysPattern, sysPatListID, 16);
    PenPat(sysPattern);
    PmForeColor(35);
    PmBackColor(229);
    FrameRect(theRect);

    OffsetRect(theRect, 245, 0);
    GetIndPattern(sysPattern, sysPatListID, 37);
    PenPat(sysPattern);
    PmForeColor(229);
    PmBackColor(210);
    PaintRect(theRect);

    OffsetRect(theRect, -245, 130);
    GetIndPattern(sysPattern, sysPatListID, 18);
    PmForeColor(210);
```

```
    PmBackColor(11);
    FillRoundRect(theRect, 50, 50, sysPattern);

    OffsetRect(theRect, 245, 0);
    GetIndPattern(sysPattern, sysPatListID, 19);
    PmForeColor(1);
    PmBackColor(0);
    FillOval(theRect, sysPattern);

    MoveTo(10, 20);
    PmForeColor(1);
    DrawString('Foreground  background colours set with PmForeColor  PmBackColor');
    NumToString(longint(a+1), theString);
    DrawString('       Palette No ');
    DrawString(theString);

    if (a = 0) then
      begin
      SetWTitle(gWindowPtr, 'Click mouse for another palette');
      while not (Button) do ;
      DisposePalette(paletteHdl);
      end;
    end;
      {of for loop}

  SetWTitle(gWindowPtr, 'Color QuickDraw');
  DisposePalette(paletteHdl);
  PenPat(qd.black);
  end;
    {of procedure DoBitPattern}

{ ####################################################################### DoPixelPattern }

procedure DoPixelPattern;

  var
  pixpat1Hdl, pixpat2Hdl : PixPatHandle;
  theRect : Rect;
  oldClipHdl, regionAHdl, regionBHdl, regionCHdl, scrollRegionHdl : RgnHandle;
  a : integer;

  begin
  RGBBackColor(gWhiteColour);
  FillRect(gWindowPtr^.portRect, qd.white);

  pixpat1Hdl := GetPixPat(rPixelPattern1);
  if (pixpat1Hdl = nil) then
    ExitToShell;
  PenPixPat(pixpat1Hdl);
  PenSize(50, 0);
  SetRect(theRect, 15, 15, 240, 280);
  FrameRect(theRect);
  SetRect(theRect, 260, 15, 485, 280);
  FillCRect(theRect, pixpat1Hdl);

  pixpat2Hdl := GetPixPat(rPixelPattern2);
  if (pixpat2Hdl = nil) then
    ExitToShell;
  BackPixPat(pixpat2Hdl);

  regionAHdl := NewRgn;
  regionBHdl := NewRgn;
  regionCHdl := NewRgn;
  SetRect(theRect, 65, 15, 190, 280);
  RectRgn(regionAHdl, theRect);
  SetRect(theRect, 260, 15, 485, 280);
  RectRgn(regionBHdl, theRect);
  UnionRgn(regionAHdl, regionBHdl, regionCHdl);

  oldClipHdl := NewRgn;
  GetClip(oldClipHdl);
  SetClip(regionCHdl);

  SetRect(theRect, 65, 15, 485, 280);

  scrollRegionHdl := NewRgn;

```

```
  for a := 0 to 279 do
    begin
    ScrollRect(theRect, 0, 1, scrollRegionHdl);
    theRect.top := theRect.top + 1;
    end;

  SetRect(theRect, 65, 15, 485, 280);
  BackPixPat(pixpat1Hdl);

  for a := 0 to 279 do
    begin
    ScrollRect(theRect, 0, -1, scrollRegionHdl);
    theRect.bottom := theRect.bottom - 1;
    end;

  SetClip(oldClipHdl);

  DisposePixPat(pixpat1Hdl);
  DisposePixPat(pixpat2Hdl);
  DisposeRgn(oldClipHdl);
  DisposeRgn(regionAHdl);
  DisposeRgn(regionBHdl);
  DisposeRgn(regionCHdl);
  DisposeRgn(scrollRegionHdl);

  PenPat(qd.black);
  end;
    {of procedure DoPixelPattern}

{ ####################################################################### DoCopyDeepMask }

procedure DoCopyDeepMask;

  var
  sourceWindowPtr : WindowPtr;
  picture1Hdl, picture2Hdl : PicHandle;
  sourceRect, maskRect, destRect, maskDisplayRect : Rect;
  windowPortPtr : CGrafPtr;
  deviceHdl : GDHandle;
  gworldPortPtr : GWorldPtr;
  gworldPixMapHdl : PixMapHandle;
  regionHdl : RgnHandle;
  finalTicks : UInt32;
  ignored : OSErr;
  alsoIgnored : boolean;

  begin
  RGBForeColor(gBlackColour);
  RGBBackColor(gWhiteColour);
  FillRect(gWindowPtr^.portRect, qd.white);

  sourceWindowPtr := GetNewCWindow(rImageWindow, nil, WindowPtr(-1));
  if (sourceWindowPtr = nil) then
    ExitToShell;
  SetPort(sourceWindowPtr);

  picture1Hdl := GetPicture(rPicture);
  if (picture1Hdl = nil) then
    ExitToShell;
  HNoPurge(Handle(picture1Hdl));
  SetRect(sourceRect, 10, 10, 167, 122);
  DrawPicture(picture1Hdl, sourceRect);
  HPurge(Handle(picture1Hdl));

  SetRect(maskRect, 0, 0, 157, 112);
  GetGWorld(windowPortPtr, deviceHdl);
  ignored := NewGWorld(gworldPortPtr, 0, maskRect, nil, nil, 0);
  SetGWorld(gworldPortPtr, nil);
  gworldPixMapHdl := GetGWorldPixMap(gworldPortPtr);
  alsoIgnored := LockPixels(gworldPixMapHdl);
  EraseRect(gworldPortPtr^.portRect);
  picture2Hdl := GetPicture(rPicture+1);
  if (picture2Hdl = nil) then
    ExitToShell;
  HNoPurge(Handle(picture2Hdl));
  DrawPicture(picture2Hdl, maskRect);
  SetGWorld(windowPortPtr, deviceHdl);
```

```

  SetPort(gWindowPtr);
  SetRect(maskDisplayRect, 19, 165, 176, 277);
  DrawPicture(picture2Hdl, maskDisplayRect);
  HPurge(Handle(picture2Hdl));
  MoveTo(43, 160);
  DrawString('Copy of offscreen mask');

  SetRect(destRect, 220, 20, 470, 275);
  regionHdl := NewRgn;
  OpenRgn;
  FrameOval(destRect);
  CloseRgn(regionHdl);

  PenSize(1, 1);
  PenPat(qd.ltGray);
  FrameRgn(regionHdl);
  MoveTo(315, 150);
  DrawString('The region');

  SetWTitle(sourceWindowPtr, 'Click Mouse to Copy');
  while not (Button) do ;
  FillRect(destRect, qd.white);

  CopyDeepMask(GrafPtr(sourceWindowPtr)^.portBits, GrafPtr(gworldPortPtr)^.portBits,
              GrafPtr(gWindowPtr)^.portBits, sourceRect, maskRect,
              destRect, srcCopy+ditherCopy, regionHdl);

  SetWTitle(sourceWindowPtr, 'Click Mouse to Close');
  Delay(60, finalTicks);

  while not (Button) do ;
  FillRect(gWindowPtr^.portRect, qd.white);

  UnlockPixels(gworldPixMapHdl);
  DisposeGWorld(gworldPortPtr);

  ReleaseResource(Handle(picture1Hdl));
  ReleaseResource(Handle(picture2Hdl));
  DisposeRgn(regionHdl);
  DisposeWindow(sourceWindowPtr);

  PenPat(qd.black);
  end;
    {of procedure DoCopyDeepMask}

{ ####################################################################### DoCheckMonitor }

function DoCheckMonitor : integer;

  var
  mainDeviceHdl : GDHandle;
  result : integer;
  alertString : string;
  pixMapHdl : PixMapHandle;
  pixelDepth : integer;
  ignored : OSErr;

  begin
  mainDeviceHdl := LMGetMainDevice;
  result := HasDepth(mainDeviceHdl, 16, 0, 0);

  if (result = 0)
    thenbegin
        GetIndString(alertString, rIndexedStrings, sNeedMonitor);
        ParamText(alertString, '', '', '');
        ignored := NoteAlert(rAlert, nil);
        DoCheckMonitor := 0;
        Exit(DoCheckMonitor);
        end

    elsebegin
        pixMapHdl := mainDeviceHdl^^.gdPMap;
        pixelDepth := pixMapHdl^^.pixelSize;
        if (pixelDepth < 16) then
          begin
          GetIndString(alertString, rIndexedStrings, sSettingMonitor);
```

```
          ParamText(alertString, '', '', '');
          ignored := NoteAlert(rAlert, nil);
          ignored := SetDepth(mainDeviceHdl, 16, 0, 0);
          DoCheckMonitor := pixelDepth;
          Exit(DoCheckMonitor);
          end;
        DoCheckMonitor := 2;
        end;
  end;
    {of function DoCheckMonitor}

{ ##################################################################### DoRestoreMonitor }

procedure DoRestoreMonitor(monitorCheckResult : integer);

  var
  alertString : string;
  mainDeviceHdl : GDHandle;
  ignored : OSErr;

  begin
  GetIndString(alertString, rIndexedStrings, sRestoringMonitor);
  ParamText(alertString, '', '', '');
  ignored := NoteAlert(rAlert, nil);

  mainDeviceHdl := LMGetMainDevice;
  ignored := SetDepth(mainDeviceHdl, monitorCheckResult, 0, 0);
  end;
    {of procedure DoRestoreMonitor}

{ ###################################################################### DoTransferModes }

procedure DoTransferModes;

  var
  monitorCheckResult, transferMode, stringIndex : integer;
  sourceWindowPtr : WindowPtr;
  sourceHdl, destinationHdl : PicHandle;
  sourceRect, destRect, blankRect : Rect;
  modeString : string;
  finalTicks : UInt32;

  begin
  monitorCheckResult := DoCheckMonitor;
  if (monitorCheckResult = 0) then
    Exit(DoTransferModes);

  RGBForeColor(gBlackColour);
  RGBBackColor(gWhiteColour);
  FillRect(gWindowPtr^.portRect, qd.white);

  sourceWindowPtr := GetNewCWindow(rImageWindow, nil, WindowPtr(-1));
  if (sourceWindowPtr = nil) then
    ExitToShell;
  SetWTitle(sourceWindowPtr, 'Source Image');

  SetPort(sourceWindowPtr);
  sourceHdl := GetPicture(rPicture);
  if (sourceHdl = nil) then
    ExitToShell;
  HNoPurge(Handle(sourceHdl));
  SetRect(sourceRect, 10, 10, 167, 122);
  DrawPicture(sourceHdl, sourceRect);
  HPurge(Handle(sourceHdl));

  SetPort(gWindowPtr);
  destinationHdl := GetPicture(rPicture+2);
  if (destinationHdl = nil) then
    ExitToShell;
  HNoPurge(Handle(destinationHdl));
  SetRect(destRect, 19, 165, 176, 277);
  DrawPicture(destinationHdl, destRect);
  MoveTo(55, 160);
  DrawString('Destination Image');

  SetRect(destRect, 270, 95, 427, 207);
  DrawPicture(destinationHdl, destRect);
```

```
  SetRect(blankRect, 270, 50, 427, 207);

  stringIndex := 5;
  for transferMode := 0 to 39 do
    begin
    if (transferMode = 8) then
      transferMode := 32;

    GetIndString(modeString, rIndexedStrings, stringIndex);
    MoveTo(270, 70);
    DrawString('Click Mouse for ');
    DrawString(modeString);

    while not (Button) do ;

    FillRect(blankRect, qd.white);
    DrawPicture(destinationHdl, destRect);
    Delay(30, finalTicks);

    CopyBits(GrafPtr(sourceWindowPtr)^.portBits, GrafPtr(gWindowPtr)^.portBits,
             sourceRect, destRect, transferMode + ditherCopy, nil);

    MoveTo(270, 92);
    if (transferMode < 8)
      thenDrawString('Boolean mode: ')
      elseDrawString('Arithmetic mode: ');
    DrawString(modeString);
    Delay(60, finalTicks);
    stringIndex := stringIndex + 1;
    end;
      {of for loop}

  MoveTo(270, 70);
  DrawString('Click Mouse to exit');
  while not (Button) do ;

  FillRect(gWindowPtr^.portRect, qd.white);

  ReleaseResource(Handle(sourceHdl));
  ReleaseResource(Handle(destinationHdl));
  DisposeWindow(sourceWindowPtr);

  if (monitorCheckResult <> 2) then
    DoRestoreMonitor(monitorCheckResult);
  end;
    {of procedure DoTransferModes}

{ ####################################################################### DoHighlighting }

procedure DoHighlighting;

  var
  grafVarsHdl : GVarHandle;
  oldHighlightColour : RGBColor;
  a : integer;
  theRect : Rect;
  hiliteVal : ByteParameter;
  finalTicks : UInt32;

  begin
  RGBBackColor(gWhiteColour);
  FillRect(gWindowPtr^.portRect, qd.white);

  grafVarsHdl := GVarHandle (CGrafPtr(gWindowPtr)^.grafVars);
  oldHighlightColour := grafVarsHdl^^.rgbHiliteColor;

  for a := 0 to 2 do
    begin
    MoveTo(20, a*80+40);
    DrawString('Clearing the highlight bit and calling InvertRect.');
    Delay(60, finalTicks);
    SetRect(theRect, 10, a * 80 + 20, 490, a * 80 + 80);

    hiliteVal := LMGetHiliteMode;
    BitClr(Ptr(@hiliteVal), pHiliteBit);
    LMSetHiliteMode(hiliteVal);

```

```
    if (a = 1)
      thenHiliteColor(gOchreColour)
      else  if (a = 2) then
            HiliteColor(gGreenColour);
    InvertRect(theRect);

    MoveTo(20, a*80+55);
    Delay(60, finalTicks);
    DrawString('Click mouse to unhighlight.  ');
    DrawString('(Note:  The call to EraseRect reset the highlight bit ...');

    while not (Button) do ;

    MoveTo(20, a*80+70);
    DrawString('... so we clear the highlight bit again before calling InvertRect.)');
    Delay(60, finalTicks);

    BitClr(Ptr(@hiliteVal), pHiliteBit);
    LMSetHiliteMode(hiliteVal);

    InvertRect(theRect);
    end;
      {of for loop}

  HiliteColor(oldHighlightColour);
  Delay(60, finalTicks);
  MoveTo(20, 260);
  DrawString('Original highlight colour has been reset.');
  end;
    {of procedure DoHighlighting}

{ ####################################################################### DoColourTable }

procedure DoColourTable;

  var
  pixMapHdl : PixMapHandle;
  colorTableHdl : CTabHandle;
  entries, a, b, c, i, j : integer;
  theRect : Rect;
  theColour : RGBColor;

  begin
  entries := 0;
  a := 0;
  b := 0;
  c := 0;
  RGBForeColor(gBlackColour);
  FillRect(gWindowPtr^.portRect, qd.black);

  pixMapHdl := CGrafPtr(gWindowPtr)^.portPixMap;
  colorTableHdl := pixMapHdl^^.pmTable;
  entries := colorTableHdl^^.ctSize;

  if (entries = 0)
    thenbegin
      RGBForeColor(gWhiteColour);
      MoveTo(90,135);
      DrawString('You need to set the monitor to 256 colours or less to get some');
      MoveTo(90,150);
      DrawString('entries in the colour table.  At present, we have zero entries.');
      end;

  for i := 0 to 15 do
    begin
    a := i * 30 + 12;
    for j := 0 to 15 do
      begin
      b := j * 18 + 5;
      if (c > entries) then Exit(DoColourTable);
      SetRect(theRect, a, b, a+28, b+17);
      theColour := colorTableHdl^^.ctTable[c].rgb;
      c := c + 1;
      RGBForeColor(theColour);
      PaintRect(theRect);
      end;
    end;
```

```
  end;
    {of procedure DoColourTable}

{ ######################################################################### DoRGBColours }

procedure DoRGBColours;

  begin
  gWhiteColour.red := $FFFF;
  gWhiteColour.green := $FFFF;
  gWhiteColour.blue := $FFFF;

  gBlackColour.red := $0000;
  gBlackColour.green := $0000;
  gBlackColour.blue := $0000;

  gOchreColour.red := $CCCC;
  gOchreColour.green := $71FC;
  gOchreColour.blue := $6A28;

  gGreenColour.red := $460D;
  gGreenColour.green := $CCCC;
  gGreenColour.blue := $6BE2;
  end;
    {of procedure DoRGBColours}

{ ################################################################## DoDemonstrationMenu }

procedure DoDemonstrationMenu(menuItem : integer);

  begin
  case (menuItem) of

    iBitPattern:
      begin
      DoBitPattern;
      end;

    iPixelPattern:
      begin
      DoPixelPattern;
      end;

    iCopyDeepMask:
      begin
      DoCopyDeepMask;
      end;

    iTransferModes:
      begin
      DoTransferModes;
      end;

    iHighlighting:
      begin
      DoHighlighting;
      end;

    iColorTable:
      begin
      DoColourTable;
      end;

    end;
      {of case statement}
  end;
    {of procedure DoDemonstrationMenu}

{ ######################################################################### DoMenuChoice }

procedure DoMenuChoice(menuChoice : longint);

  var
  menuID, menuItem : integer;
  itemName : string;
  daDriverRefNum : integer;

```

```
  begin
  menuID := HiWord(menuChoice);
  menuItem := LoWord(menuChoice);

  if (menuID = 0) then
    Exit(DoMenuChoice);

  case (menuID) of

    mApple:
      begin
      GetMenuItemText(GetMenuHandle(mApple), menuItem, itemName);
      daDriverRefNum := OpenDeskAcc(itemName);
      end;

    mFile:
      begin
      if (menuItem = iQuit) then
        gDone := true;
      end;

    mDemonstration:
      begin
      DoDemonstrationMenu(menuItem);
      end;

    end;
      {of case statement}

  HiliteMenu(0);
  end;
    {of procedure DoMenuChoice}

{ ########################################################################## DoMouseDown }

procedure DoMouseDown(var eventRec : EventRecord);

  var
  myWindowPtr : WindowPtr;
  partCode : integer;

  begin
  partCode := FindWindow(eventRec.where, myWindowPtr);

  case (partCode) of

    inMenuBar:
      begin
      DoMenuChoice(MenuSelect(eventRec.where));
      end;

    inSysWindow:
      begin
      SystemClick(eventRec, myWindowPtr);
      end;

    inContent:
      begin
      if (myWindowPtr <> FrontWindow) then
        SelectWindow(myWindowPtr);
      end;

    inDrag:
      begin
      DragWindow(myWindowPtr, eventRec.where, qd.screenBits.bounds);
      end;

    inGoAway:
      begin
      if (TrackGoAway(myWindowPtr, eventRec.where)) then
        gDone := true;
      end;

    end;
      {of case statement}
  end;
    {of procedure DoMouseDown}
```

```
{ ############################################################################# DoEvents }

procedure DoEvents(var eventRec : EventRecord);

  var
  myWindowPtr : WindowPtr;
  charCode : char;

  begin
  myWindowPtr := WindowPtr(eventRec.message);

  case (eventRec.what) of

    mouseDown:
      begin
      DoMouseDown(eventRec);
      end;

    keyDown, autoKey:
      begin
      charCode := chr(BAnd(eventRec.message, charCodeMask));
      if (BAnd(eventRec.modifiers, cmdKey) <> 0) then
        DoMenuChoice(MenuKey(charCode));
      end;

    updateEvt:
      begin
      BeginUpdate(myWindowPtr);
      EndUpdate(myWindowPtr);
      end;

    end;
      {of case statement}
  end;
    {of procedure DoEvents}

{ ################################################################# start of main program }

begin

  { ………………………………………………………………………………………………………………………………………………………… initialise managers }

  DoInitManagers;

  { …………………………………………………………………………………………………………………………………………………… check for Color QuickDraw }

  theErr := Gestalt(gestaltQuickdrawVersion, response);
  if (response < gestalt8BitQD) then
    begin
    GetIndString(alertString, rIndexedStrings, sColorQuickdraw);
    ParamText(alertString, '', '', '');
    ignored := StopAlert(rAlert, nil);
    ExitToShell;
    end;

  { …………………………………………………………………………………………………………………………………………………… set up menu bar and menus }

  menubarHdl := GetNewMBar(rMenubar);
  if (menubarHdl = nil) then
    ExitToShell;
  SetMenuBar(menubarHdl);
  DrawMenuBar;

  menuHdl := GetMenuHandle(mApple);
  if (menuHdl = nil)
    thenExitToShell
    elseAppendResMenu(menuHdl, 'DRVR');

  { ……………………………………………………………………………………………………………………………………………………………………… open window }

  gWindowPtr := GetNewCWindow(rWindow, nil, WindowPtr(-1));
  if (gWindowPtr = nil) then
    ExitToShell;

  SetPort(gWindowPtr);
  TextSize(10);
```

```
845
846     { ………………………………………………………………………………………………………………………………………… create some RGB colours }
847
848     DoRGBColours;
849
850     { ……………………………………………………………………………………………………………………………………………………………… eventLoop }
851
852     gDone := false;
853
854     while not (gDone) do
855       begin
856       gotEvent := WaitNextEvent(everyEvent, eventRec, kMaxLong, nil);
857       if (gotEvent) then
858         DoEvents(eventRec);
859       end;
860
861   end.
862
863   { ####################################################################################### }
```

Demonstration Program Comments

When this program is run, the user should invoke demonstrations of various Color QuickDraw drawing operations by choosing items from the Demonstration menu. One demonstration (Transfer Modes) will not be invoked unless the user's machine is capable of displaying at least 16-bit colour.

The constant declaration block

Lines 50-59 establish constants related to menu IDs and item numbers. Lines 60-68 establish constants related to resource IDs. The constants at Lines 69-72 are used to index the 'STR#' resource. Line 74 defines kMaxLong as the maximum possible long value. This value will be assigned to WaitNextEvent's sleep parameter.

The variable declaration block

gDone controls program termination. gWindowPtr will be assigned a pointer to the main window. The remaining globals will be assigned RGB colour values for black, white, ochre and green.

The procedure DoBitPattern

DoBitPattern is the first demonstration. It demonstrates the use of bit patterns in Color QuickDraw. It also demonstrates the use of palettes and Palette Manager routines to specify colours.

Note that, as is the case with all drawing demonstration functions in this program, some of the code is related to program execution (for example, delays, setting the window title, waiting for mouse clicks before proceeding, etc) and not to drawing operations per se. Those parts of the code will generally be disregarded in the following comments.

Line 126 initiates a loop which will cycle twice. The first time through, some shapes will be drawn using one palette's colours. The second time through, the same shapes will be drawn using the same colour index numbers, but with another palette.

Line 128 retrieves a palette from a 'pltt' resource, allocating and initialising a new Palette data structure. Line 129 applies that palette's values to the specified window.

Line 131 uses the Palette Manager routine PmBackColor to set the background colour, and Line 131 fills the port rectangle with that colour.

Lines 136-140 retrieve one of the system patterns, set the pen pattern to that pattern, set the foreground and background colours to particular values, and draw a framed rectangle. Lines 142-159 change the pen pattern and colours between painting a rectangle, filling a round rectangle and filling an oval.

At Line 172, and during the first passage through the loop only, the memory occupied by the Palette data structure is deallocated. When the loop repeats, a second palette's values will be applied to the window (Lines 128-129). The memory occupied by the second Palette data structure is deallocated at Line 178.

The procedure DoPixelPattern

DoPixelPattern demonstrates pixel patterns. A framed and a filled rectangle are drawn. The ScrollRect routine is then used to scroll the foreground out of the rectangles, replacing the scrolled areas with a background pixel pattern, the drawing associated with the scrolling being restricted by a clipping region comprising two separate rectangles.

Lines 194-195 set all pixels in the port rectangle to white.

At Line 197-200, a pixel pattern is retrieved from a 'ppat' resource and assigned to the pen. A framed rectangle is then drawn (Line 203). Note that the pen height is set to zero (Line 201), meaning that the two sides of the rectangle will be drawn but not the top and bottom.

Lines 204-205 draw a filled rectangle using the retrieved pixel pattern. Note that, under Color QuickDraw, the FillCRect, not the FillRect routine is used.

At Line 207, a new pixel pattern is retrieved from another 'ppat' resource. At Line 210, this new pixel pattern becomes the background pattern.

Lines 212-219 create a region comprising two separate rectangles (one coincident with the "inside" of the framed rectangle and the other coincident with the whole of the filled rectangle). The current clipping region is then saved and the newly created region is established as the clipping region (Lines 221-223).

Line 225 establishes a rectangle for the ScrollRect routine. Laterally, this extends from the left inside of the framed rectangle to the right hand side of the filled rectangle. Line 227 creates the empty region required by the ScrollRect call.

Lines 229-233 scroll the rectangle downwards, the top of the rectangle being incremented downwards between calls to ScrollRect. ScrollRect fills the "vacated" areas within the clipping region with the background pattern .

Lines 235-236 reset the rectangle and change the background pattern to the first pattern. The scrolling operation is then repeated, this time in an upwards direction (Lines 238-242).

Line 244 resets the clipping region to the region saved at Line 222. Lines 246-252 deallocate the memory associated with the pixel patterns and regions.

The procedure DoCopyDeepMask

DoCopyDeepMask demonstrates the CopyDeepMask routine. A 16-bit source picture in one pixel map is copied through a 16-bit mask in another (offscreen) pixel map to a destination. The resulting image is scaled up and clipped to an oval-shaped region.

Firstly, at Lines 276-277, the foreground and background colours are set to black and white respectively. (This should always be done before a call to CopyBits, CopyMask or CopyDeepMask.) The window's port is then cleared to white.

Line 280 opens a small window, which will be used for the source image. Lines 283-291 set the current port to this window's port, retrieve the source picture from a 'PICT' resource and draw the picture in the window. (Since the 'PICT' resource has the purgeable bit set, it is made non-purgeable immediately after it is retrieved, used immediately, and made purgeable immediately after it is used.)

The mask pixel map cannot come from the screen. Accordingly, Lines 293-305 create an offscreen graphics world, retrieve from a resource the picture to be used as the mask, and draw the picture in the offscreen graphics port. (Note: Offscreen graphics world are addressed at Chapter 12 - Offscreen Graphics Worlds, Pictures, Cursors and Icons. The code here is "bare bones" and does not check for errors.)

Lines 307-310 set the drawing graphics port to the main window, draws the mask image in the main window (simply so that the user can see what it looks like) and makes the associated 'PICT' resource purgeable now that it has been used for the last time.

Lines 314-318 create an oval-shaped region. So that the user can see this otherwise invisible region, its outline is drawn in the main window at Lines 320-324.

When the user clicks the mouse button (Line 327), CopyDeepMask is called (Line 330). Note the coercion to a GrafPtr in the first three parameters, the source mode specified (srcCopy + ditherCopy) and the region specified in the last parameter.

When the user again clicks the mouse button (Line 337), the window is cleared to white, the offscreen graphics world is disposed of (Lines 340-341), memory associated with the pictures and the region is deallocated (Lines 341-345), and the small source window is disposed of (Line 346).

The procedure DoCheckMonitor

DoCheckMonitor checks if the user's main display device can display at least 16-bit direct colour. If it cannot, the function informs the user via a dialog box and returns. If it can, but if the pixel depth is currently set to a value lower than 16, the pixel depth is set to 16 after the user is informed of this imminent action via an Alert box. If the pixel depth is currently at least 16, the function simply returns.

Line 365 gets a handle to the startup device. Line 366 checks whether the device supports a pixel depth of 16. Lines 368-375 deal with the case of a device which does not support direct colour.

The next step, if we are dealing with a direct device (Line 377), is to check the current pixel depth setting. The method used here is to extract this value from the pixelSize field of the PixMap record (Lines 378-379). If the value is less than 16 (Line 380), an advisory Alert box is called up (Lines 382-384), SetDepth is called at Line 385 to set the device to a pixel depth of 16, and the old pixel depth is returned to the calling function (Line 386).

If the pixel depth is already at least 16, Line 389 simply returns a positive value to the calling function, no action having been taken by DoCheckMonitor.

The procedure DoRestoreMonitor

If DoCheckMonitor changed the pixel depth of the user's display device, DoRestoreMonitor is called to return that device to the pixel depth setting prior to DoCheckMonitor being called. This value is passed to DoRestoreMonitor as a formal parameter, having been passed to the calling function at Line 386 of the DoCheckMonitor function.

Lines 404-406 first notify the user of the intended action via an Alert box. Lines 408-410 effect the change.

The procedure DoTransferModes

DoTransferModes demonstrates the Boolean and arithmetic transfer modes. At each click of the user's mouse, a 16-bit source image is copied from one graphics port to another, overwriting an image in the destination port. Each time the image is copied (using CopyBits), the transfer mode is changed.

Firstly, at Line 426, a check is made of the user's display device. If the device is not capable of displaying at least 16-bit colour, the function is exited (Line 428) following DoCheckMonitor's advice to the user via an Alert box. If the device is capable of displaying at least 16-bit colour, but is currently set to 256 colours or less, DoCheckMonitor will reset the device's pixel depth to 16, advising the user of this action via an Alert box.

Since CopyBits will be called, Lines 430-431 set the foreground and background colours to black and white respectively. Line 432 clears the window to white.

Line 434 opens a small window for the source image. Lines 440-445 retrieve a picture from a 'PICT' resource and draw the picture in the small window. (Since the 'PICT' resource is purgeable, it is made non-purgeable immediately it is retrieved, used immediately, and immediately made purgeable again.) Lines 449-454 retrieve another picture from a 'PICT' resource and draw it into the bottom left of the main window. Lines 458-459 draw the same picture in the right-middle of the main window. (The first draw is simply to continually display to the user the appearance of the "destination" image. The second draw is the actual destination to which the source pixel image will be copied.)

Line 462 establishes a loop which will be traversed once for each of the Boolean and arithmetic transfer modes, with each traverse being initiated by the user clicking the mouse button. The name of the transfer mode invoked during each traverse is printed in the window.

When the user clicks the mouse button (Line 473), the destination image is re-drawn in the right-middle of the display window (Line 476). CopyBits is then called at Line 479 to copy the source pixel image to the destination. Note that the transfer mode specified in this call is changed every time around the loop.

When the loop exits and the user responds to a request for a terminating click of the mouse button (Lines 493-494), the port rectangle is filled with the background colour (Line 496), memory associated with the pictures is deallocated (Lines 498-499), and the small window is disposed of (Line 500).

If Line 426 resulted in the program resetting the device's pixel depth, Lines 502-503 reset the device to the old pixel depth saved at Line 426.

The procedure DoHighlighting

DoHighlighting demonstrates highlighting, first with the colour set by the user in the Colour control panel, and then with two colours set by the program.

Firstly, at Lines 523-524, the current highlight colour is saved.

Line 526 then initiates a loop which will be traversed three times. On the second and third traverses, the highlight colour will be changed.

Within the loop, at Lines 533-535, a copy of the value at the low memory global HiliteMode is acquired, BitClr is called to clear the highlight bit, and HiliteMode is set to this new value. At Lines 539-540, the highlight colour is changed if this is the second or third time around the loop. With the highlight bit cleared, InvertRect is called at Line 541 to invert a specified rectangle.

Note that the call to InvertRect (Line 541) resets the highlight bit. Accordingly, when the user clicks the mouse button (Line 548), the highlight bit is cleared once again (Lines 554-555) before InvertRect is called once again. This second call restores the colour in the specified rectangle to the background colour.

Before the DoHighLighting function returns, it sets the highlight colour (Line 561) to the original highlight colour saved at Line 524.

The procedure DoColourTable

DoColourTable draws small rectangles in the window, one for each of the entries in the current colour table. The trail to those entries, which are stored in an array, is from the CGrafPort record to the PixMap record to the ColorTable record, and thence to each of the ColorSpec records in the ctTable field (an array of type CSpecArray) of that ColorTable record.

Note that there will be no entries in the colour table unless the device has been set to 256 colours or less at some time during the current session.

Line 587 retrieves the handle to the PixMap record from the colour graphics port record. Line 589 gets the handle to the ColorTable record. Line 589 retrieves the number of entries in the colour table.

If the colour table contains no entries (Line 591), a message is drawn in the window advising the user that the monitor needs to be set to 256 colours or lessin order to view a colour table (Lines 592-598).

The loop entered at Line 600 draws a rectangle for each array element in the ctTable field of the ColorTable record. The variable c, which is incremented each time around the loop until it is greater than the number of colour table entries, controls the exit from loop (Line 606). The variable c also controls which RGBColor entry in the colour table is assigned as the foreground colour each time through the loop (Lines 609-610).

The procedure DoRGBColours

DoRGBColours assigns colours to the global variables declared at Lines 82-85.

The procedures DoDemonstrationMenu, DoMenuChoice

DoMenuChoice and DoDemonstrationMenu handle menu choices from the Apple, File and Demonstration menus.

The procedures DoMouseDown, DoEvents

DoEvents and DoMouseDown perform minimal event handling consistent with the satisfactory operation of the drawing aspects of the demonstration.

The main program block

The main function initialises the system software managers (Line 811) and then checks whether the Color QuickDraw is available (Lines 815-816). If it is not, Lines 818-821 invoke a Stop alert advising the user that the program requires Color QuickDraw. When the user clicks the OK button, the program terminates.

Lines 826-835 set up the menus.

Line 839 opens the main window. Since GetNewCWindow is used, the window will be created with a colour graphics port.

Line 843 sets this window's colour graphics port as the current port for drawing and Line 844 sets the text size to 10 points.

Line 848 calls the application-defined routine doRGBColours to assign colour values to the global variables declared at Lines 82-85.

The main event loop is entered at Line 854. It terminates when gDone is set to true.

Note that here, as in other areas of the program, error handling is somewhat rudimentary: the program simply terminates.

Creating `'pltt'` and `'ppat'` Resources Using ResEdit

Creating `'pltt'` Resources

The procedure for creating the two `'pltt'` resources is as follows:

- Open BasicQuickDraw.µ.rsrc in ResEdit. Choose **Resource/Create New Resource**. A small dialog opens. Click the **pltt** item in the scrolling list, and then click the dialog's **OK** button. The **pltts from ColorQuickDraw.µ.rsrc** window opens, followed by the **pltt ID = 128 from ColorQuickDraw.µ.rsrc** window. (ResEdit automatically assigns 128 as the resource ID of the first `'pltt'` resource you create.) Note that the palette is currently empty.

- Choose **pltt/Load Colors…**. A dialog opens. Click on the **pltt** radio button. Click on the items in the list to explore the palettes. Click on the item **ResEdit Standard Colors** and click the **OK** button. The dialog closes and the palette appears in the **pltt ID = 128 from ColorQuickDraw.µ.rsrc** window. Before clicking the go-away box to close that window, note the following:

  - When you click a single colour patch, you can change its value by typing new numbers into the **Red**, **Green**, and **Blue** editable text items, or by clicking the up and down arrows.

  - You can create a colour ramp by Shift-clicking two colour patches to create a selection and then choosing **pltt/Blend**.

  - Other **pltt** menu items enable you to complement a colour and change the colour model from Red/Green/Blue to Cyan/Magenta/Yellow, Hue/Saturation/Brightness, or Hue/Lightness/Saturation.[4]

  - **Resource** menu items are available for inserting a new colour and opening the colour picker. **Background** menu items enable you to change the background to black, white, or gray.

- Click the go-away box to close the **pltt ID = 128 from ColorQuickDraw.µ.rsrc** window. Choose **Resource/Create New Resource**. The **pltt ID = 129 from ColorQuickDraw.µ.rsrc** window opens. (ResEdit automatically increments the resource ID.)

- Choose **pltt/Load Colors…**. A dialog opens. This time, click on the **clut** radio button.[5] Click on the items in the list to explore the cluts. Click on the first item **Unnamed** and click the **OK** button. A dialog appears advising you that `'pltt'` resources must always have white and black as the first two entries. Click the **OK** button. The dialog closes and the palette appears in the **pltt ID = 129 from ColorQuickDraw.µ.rsrc** window.

---

[4] Colour models are explained at Chapter 22 — Miscellany.

[5] `'clut'` and `'pltt'` resources are largely interchangeable, but the `'pltt'` resource also contains usage information. Palettes are associated with windows.

- Close the the **pltt ID = 129 from ColorQuickDraw.µ.rsrc** window. Close the **pltts from ColorQuickDraw.µ.rsrc** window. A pltt icon representing the resources just created appears in the **ColorQuickDraw.µ.rsrc** window.

Creating `'ppat'` Resources

The procedure for creating the two `'ppat'` resources is as follows:

- Choose **Resource/Create New Resource**. A small dialog opens. Click the **ppat** item in the scrolling list, and then click the dialog's **OK** button. The **ppats from ColorQuickDraw.µ.rsrc** window opened, followed by the **ppat ID = 128 from ColorQuickDraw.µ.rsrc** window. (ResEdit automatically assigns 128 as the resource ID of the first `'ppat'` resource you create.)

- Choose **ppat/Pattern Size…** . In the resulting dialog, click on the box representing the desired pixel pattern size, then click the **Resize** button.

- Back in the **ppat ID = 128 from ColorQuickDraw.µ.rsrc** window, use the drawing tools provided to draw the pixel pattern in the centre box. Then close the **ppat ID = 128 from ColorQuickDraw.µ.rsrc** window.

- Choose **Resource/Create New Resource**. The **ppat ID = 129 from ColorQuickDraw.µ.rsrc** window opens. (ResEdit automatically increments the resource ID.) Repeat the previous two steps to create the pixel pattern, then close the **ppat ID = 129 from ColorQuickDraw.µ.rsrc** window.

Close the **ppats from ColorQuickDraw.µ.rsrc** window. A pltt icon representing the resources just created appears in the **ColorQuickDraw.µ.rsrc** window. Close the **ColorQuickDraw.µ.rsrc** window, saving ColorQuickDraw.µ.rsrc.