This chapter describes how Color QuickDraw manages video devices so that your application can draw to a window's graphics port without regard to the capabilities of the screen—even if the window spans more than one screen.

Read this chapter to learn how Color 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 record of data type `GDevice`. Your application generally never needs to create `GDevice` records. However, your application may find it useful to examine `GDevice` records to determine the capabilities of the user's screens. When zooming a window, for example, your application can use `GDevice` records to determine which screen contains the largest area of a window, and then determine the ideal window size for that screen. You may also wish to use the `DeviceLoop` procedure, described in this chapter, if you want to optimize your application's drawing for screens with different capabilities.

This chapter describes the `GDevice` record and the routines that Color QuickDraw uses to create and manage such records. This chapter also describes routines that your application might find helpful for determining screen characteristics. For many applications, QuickDraw provides a device-independent interface; as described in other chapters of this book, your application can draw images in a graphics port for a window, and Color QuickDraw automatically manages the path to the screen—even if the user has multiple screens. However, if your application needs more control over how it draws images on screens of various sizes and with different capabilities, your application can use the routines described in this chapter.

About Graphics Devices

A ***graphics device*** is anything into which QuickDraw can draw. There are three types of graphics devices: video devices (such as plug-in video cards and built-in video interfaces) that control screens, offscreen graphics worlds (which allow your application to build complex images off the screen before displaying them), and printing graphics ports for printers. The chapter "Offscreen Graphics Worlds" in this book describes how to use QuickDraw to draw into an offscreen graphics world; the chapter "Printing Manager" in this book describes how to use QuickDraw to draw into a printing graphics port.

For a video device or an offscreen graphics world, Color QuickDraw stores state information in a **`GDevice`** ***record.*** Note that printers do not have `GDevice` records. Color QuickDraw automatically creates `GDevice` records. (Basic QuickDraw does not create `GDevice` records, nor does basic QuickDraw support multiple screens.)

When the system starts up, it allocates and initializes a handle to a `GDevice` record for each video device it finds. When you use the `NewGWorld` function (described in the chapter "Offscreen Graphics Worlds" in this book), Color QuickDraw automatically creates a `GDevice` record for the new offscreen graphics world.

All existing `GDevice` records are linked together in a list, called the ***device list;*** the global variable `DeviceList` holds a handle to the first record in the list. At any given time, exactly one graphics device is the ***current device*** (also called the *active device*)—the one on which drawing is actually taking place. A handle to its `GDevice` record is stored in the global variable `TheGDevice`. By default, the `GDevice` record corresponding to the first video device found is marked as the current device; all other graphics devices in the list are initially marked as inactive.

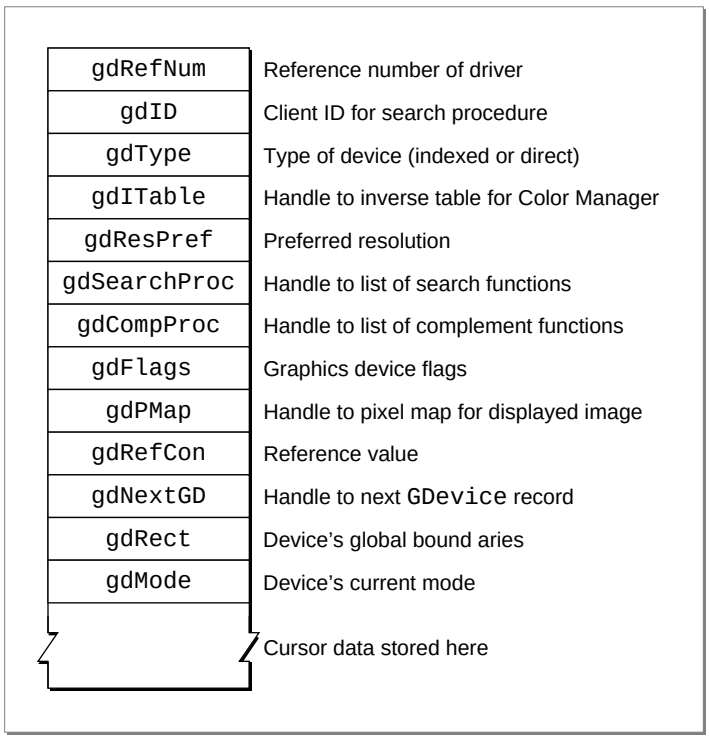
When the user moves a window or creates a window on another screen, and your application draws into that window, QuickDraw automatically makes the video device for that screen the current device. Color QuickDraw stores that information in the global variable `TheGDevice`. As Color QuickDraw draws across a user's video devices, it keeps switching to the `GDevice` record for the video device on which Color QuickDraw is actively drawing.

The user can use the Monitors control panel to set the desired pixel depth of each video device; to set the display to color, grayscale, or black and white; and to set the position of each screen relative to the ***main screen*** (that is, the one that contains the menu bar). The Monitors control panel stores all configuration information for a multiscreen system in the System file in a resource of type `'scrn'` that has a resource ID of 0. Your application should never create this resource, and should never alter or examine it. The `'scrn'` resource consists of an array of data structures that are analogous to `GDevice` records. Each element of this array contains information about a different video device.

When the `InitGraf` procedure (described in the chapter "Basic QuickDraw" in this book) initializes QuickDraw, it checks the System file for the `'scrn'` resource. If the `'scrn'` resource is found and it matches the hardware, `InitGraf` organizes the video devices according to the contents of this resource; if not, then QuickDraw uses only the video device for the startup screen.

The `GDevice` record is diagrammed in Figure 5-1. Some aspects of its contents are discussed after the figure; see page 5-15 for a complete description of the fields. Your application can use the routines described in this chapter to manipulate values for the fields in this record.

Figure 5-1 The `GDevice` record



The `gdITable` field points to an inverse table, which the Color Manager creates and maintains. An ***inverse table*** is a special Color Manager data structure arranged in such a manner that, given an arbitrary RGB color, its pixel value (that is, its index number in the CLUT) can be found quickly. The process is very fast once the table is built, but, if a color is changed in the video device's CLUT, the Color Manager must rebuild the inverse table the next time it has to find a color. The Color Manager is described in the chapter "Color Manager" in *Inside Macintosh: Advanced Color Imaging*.

The `gdPMap` field contains a handle to the pixel map that reflects the imaging capabilities of the graphics device. The pixel map's `PixelType` and `PixelSize` fields indicate whether the graphics device is direct or indexed and what pixel depth it displays. Color QuickDraw automatically synchronizes this pixel map's color table with the CLUT on the video device.

The `gdRect` field describes the graphics device's boundary rectangle in global coordinates. Color QuickDraw maps the (0,0) origin point of the global coordinate plane to the main screen's upper-left corner, and other screens are positioned adjacent to the main screen according to the settings made by the user with the Monitors control panel.

Using Graphics Devices

To use graphics devices, your application generally uses the QuickDraw routines described elsewhere in this book to draw images into a window; Color QuickDraw automatically displays your images in a manner appropriate for each graphics device that contains a portion of that window.

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

Instead of drawing directly into an onscreen graphics port, your application can use an offscreen graphics world (described in the chapter "Offscreen Graphics Worlds") to create images with the ideal pixel depth and color table required by your application. Then your application can use the `CopyBits` procedure to copy the images to the screen. Color QuickDraw converts the colors of the images for appropriate display on grayscale graphics devices and on direct and indirect color graphics devices. The manner in which Color QuickDraw translates the colors specified by your application to different graphics devices is described in the chapter "Color QuickDraw." However, if Color QuickDraw were to translate the colors of a color wheel (such as that used by the Color Picker, described in *Inside Macintosh: Advanced Color Imaging*), the image would appear as solid black on a black-and-white screen.

Many applications can let Color QuickDraw manage multiple video devices of differing dimensions and pixel depths. If your application needs more control over video device management—if it needs certain pixel depths or sets of colors to function effectively, for example—you can take several steps.

- If you need to know about the characteristics of available video devices, your application can use the `GetDeviceList` function to obtain a handle to the first `GDevice` record in the device list, the `GetGDevice` function to obtain a handle to the `GDevice` record for the current device, the `GetMainDevice` function to obtain a handle to the `GDevice` record for the main screen, or the `GetMaxDevice` function to obtain a handle to the `GDevice` record for the graphics device with the greatest pixel depth. Your application can then pass this handle to a routine like the `TestDeviceAttribute` function or the `HasDepth` function to determine various characteristics of a video device, or your application can examine the `gdRect` field of the `GDevice` record to determine the dimensions of the screen it represents.
- If you want to optimize your application's drawing for the best possible display on whatever type of screen is the current device, your application can use the `DeviceLoop` procedure, described on page 5-29, to determine the capabilities of the current device before drawing into a window on that device.
- If the current device is not suitable for the proper display of an image—for example, if the user has moved the window for your multicolored display of national flags to a black-and-white screen—your application can display the best image possible and display a message explaining that a more capable screen is required for better presentation of the image. Your application can use the `DeviceLoop` procedure to determine the capabilities of the current device.
- If your application uses the `HasDepth` function to determine that the current device can support the pixel depth required for the proper display of your image, but the `DeviceLoop` procedure indicates that the user has changed the screen's display, your application can use the `SetDepth` function to change the pixel depth of the screen. Note that the `SetDepth` function is provided for applications that are able to run only on graphics devices of a particular depth. Your application should use it only after soliciting the user's permission with a dialog box.
- If your application needs more control over colors on different indexed devices, your application can use the Palette Manager to arrange different sets of colors for particular images. Because the CLUT is variable on most video devices, your application can display up to 16 million colors, although on an 8-bit indexed device, for example, only 256 different colors can appear at once. See the chapter "Palette Manager" in *Inside Macintosh: Advanced Color Imaging* for more information.
- If your application needs to work with offscreen images that have characteristics different from those on the available graphics devices, your application can create offscreen graphics worlds, which contain their own `GDevice` records. See the chapter "Offscreen Graphics Worlds" in this book for more information.

To use the routines described in this chapter, your application must check for the existence of Color QuickDraw by using the `Gestalt` function with the `gestaltQuickDrawVersion` selector. The `Gestalt` function returns a 4-byte value in its `response` parameter; the low-order word contains QuickDraw version data. In that low-order word, the high-order byte gives the major revision number and the low-order byte gives the minor revision number. If the value returned in the `response` parameter is greater than or equal to the value of the constant `gestalt32BitQD`, then the system supports Color QuickDraw and all of the routines described in this chapter.

Optimizing Your Images for Different Graphics Devices

The `DeviceLoop` procedure searches for graphics devices that intersect your window's drawing region, and it informs your application of each different graphics device it finds. The `DeviceLoop` procedure provides your application with information about the current device's pixel depth and other attributes. Your application can then choose what drawing technique to use for the current device. For example, your application might use inversion to achieve a highlighting effect on a 1-bit graphics device, and, by using the `HiliteColor` procedure described in the chapter "Color QuickDraw," it might specify a color like magenta as the highlight color on a color graphics device.

For example, you can call `DeviceLoop` after calling the Event Manager procedure `BeginUpdate` whenever your application needs to draw into a window, as shown in Listing 5-1.

Listing 5-1 Using the `DeviceLoop` procedure

```
PROCEDURE DoUpdate (window: WindowPtr);
VAR
   windowType := Integer;
   myWindow: LongInt;
BEGIN
   windowType := MyGetWindowType(window);
   CASE windowType OF
   kSimpleRectanglesWindow: {simple case: window with 2 color rectangles}
      BEGIN
         BeginUpdate(window);
         myWindow := LongInt(window); {coerce window ptr for MyDrawingProc}
         DeviceLoop(window^.visRgn, @MyTrivialDrawingProc,
                    myWindow, []);
         EndUpdate;
      END;
   {handle other window types--documents, dialog boxes, etc.--here}
END;
```

When you use the `DeviceLoop` procedure, you must supply a handle to a drawing region and a pointer to your own application-defined drawing procedure. In Listing 5-1, a handle to the window's visible region and a pointer to an application-defined drawing procedure called `MyTrivialDrawingProc` are passed to `DeviceLoop`. For each graphics device it finds as the application updates its window, `DeviceLoop` calls `MyTrivialDrawingProc`.

Because `DeviceLoop` provides your drawing procedure with the pixel depth of the current device (along with other attributes passed to your drawing procedure in the `deviceFlags` parameter), your drawing procedure can optimize its drawing for whatever type of video device is the current device, as illustrated in Listing 5-2.

Listing 5-2 Drawing into different screens

```
PROCEDURE MyTrivialDrawingProc (depth: Integer;
                                deviceFlags: Integer;
                                targetDevice: GDHandle;
                                userData: LongInt);
VAR
   window: WindowPtr;
BEGIN
   window:= WindowPtr(userData);
   EraseRect(window^.portRect);
   CASE depth OF
   1:                              {black-and-white screen}
      MyDraw1BitRects(window);     {draw with ltGray, dkGray pats}
   2:
      MyDraw2BitRects(window); {draw with 2 of 4 available colors}
   {handle other screen depths here}
END;
```

Zooming Windows on Multiscreen Systems

The zoom box in the upper-right corner of the standard document window allows the user to alternate quickly between two window positions and sizes: the user state and the standard state.

The ***user state*** is the window size and location established by the user. If your application does not supply an initial user state, the user state is simply the size and location of the window when it was created, until the user resizes it.

The ***standard state*** is the window size and location that your application considers most convenient, considering the function of the document and the screen space available. In a word-processing application, for example, a standard-state window might show a full page, if possible, or a page of full width and as much length as fits on the screen. If the user changes the page size with the Page Setup command, the application might

adjust the standard state to reflect the new page size. If your application does not define a standard state, the Window Manager automatically sets the standard state to the entire gray region on the main screen, minus a three-pixel border on all sides. (See *Macintosh Human Interface Guidelines* for a detailed description of how your application determines where to open and zoom windows.) The user cannot change a window's standard state. (The user and standard states are stored in a data structure of type `WStateData` whose handle appears in the `dataHandle` field of the window record.)

Listing 5-3 illustrates an application-defined procedure, `DoZoomWindow`, which an application might call when the user clicks the zoom box. Because the user might have moved the window to a different screen since it was last zoomed, the procedure first determines which screen contains the largest area of the window and then calculates the ideal window size for that screen before zooming the window.

The screen calculations in the `DoZoomWindow` procedure compare `GDevice` records stored in the device list. (If Color QuickDraw is not available, `DoZoomWindow` assumes that it's running on a computer with a single screen.)

Listing 5-3 Zooming a window

```
PROCEDURE DoZoomWindow (thisWindow: windowPtr; zoomInOrOut: Integer);
VAR
   gdNthDevice, gdZoomOnThisDevice: GDHandle;
   savePort:                        GrafPtr;
   windRect, zoomRect, theSect:     Rect;
   sectArea, greatestArea:          LongInt;
   wTitleHeight:                    Integer;
   sectFlag:                        Boolean;
BEGIN
   GetPort(savePort);
   SetPort(thisWindow);
   EraseRect(thisWindow^.portRect);    {erase to avoid flicker}
   IF zoomInOrOut = inZoomOut THEN     {zooming to standard state}
   BEGIN
      IF NOT gColorQDAvailable THEN    {assume a single screen and }
      BEGIN                            { set standard state to full screen}
         zoomRect := screenBits.bounds;
         InsetRect(zoomRect, 4, 4);
         WStateDataHandle(WindowPeek(thisWindow)^.dataHandle)^^.stdState
                                                              := zoomRect;
      END
      ELSE                    {locate window on available screens}
      BEGIN
         windRect := thisWindow^.portRect;
         LocalToGlobal(windRect.topLeft);    {convert to global coordinates}
```

```
LocalToGlobal(windRect.botRight);
{calculate height of window's title bar}
wTitleHeight := windRect.top - 1 -
            WindowPeek(thisWindow)^.strucRgn^^.rgnBBox.top;
windRect.top := windRect.top - wTitleHeight;
gdNthDevice := GetDeviceList;    {get the first screen}
greatestArea := 0;         {initialize area to 0}
{check window against all gdRects in gDevice list and remember }
{ which gdRect contains largest area of window}
WHILE gdNthDevice <> NIL DO
IF TestDeviceAttribute(gdNthDevice, screenDevice) THEN
   IF TestDeviceAttribute(gdNthDevice, screenActive) THEN
   BEGIN
      {The SectRect function calculates the intersection }
      { of the window rectangle and this GDevice's boundary }
      { rectangle and returns TRUE if the rectangles intersect, }
      { FALSE if they don't.}
      sectFlag := SectRect(windRect, gdNthDevice^^.gdRect,
                           theSect);
      {determine which screen holds greatest window area}
      {first, calculate area of rectangle on current screen}
      WITH theSect DO
         sectArea := LongInt(right - left) * (bottom - top);
      IF sectArea > greatestArea THEN
      BEGIN
         greatestArea := sectArea;  {set greatest area so far}
         gdZoomOnThisDevice := gdNthDevice;  {set zoom device}
      END;
      gdNthDevice := GetNextDevice(gdNthDevice);   {get next }
   END;  {of WHILE}                                { GDevice record}
{if gdZoomOnThisDevice is on main device, allow for menu bar height}
IF gdZoomOnThisDevice = GetMainDevice THEN
   wTitleHeight := wTitleHeight + GetMBarHeight;
WITH gdZoomOnThisDevice^^.gdRect DO    {create the zoom rectangle}
BEGIN
   {set the zoom rectangle to the full screen, minus window title }
   { height (and menu bar height if necessary), inset by 3 pixels}
   SetRect(zoomRect, left + 3, top + wTitleHeight + 3,
          right - 3, bottom - 3);
   {If your application has a different "most useful" standard }
   { state, then size the zoom window accordingly.}
```

```
            {set up the WStateData record for this window}
            WStateDataHandle(WindowPeek(thisWindow)^.dataHandle)^^.stdState
                                                              := zoomRect;
         END;
      END;
   END; {of inZoomOut}
   {if zoomInOrOut = inZoomIn, just let ZoomWindow zoom to user state}
   {zoom the window frame}
   ZoomWindow(thisWindow, zoomInOrOut, (thisWindow = FrontWindow));
   MyResizeWindow(thisWindow);   {application-defined window-sizing routine}
   SetPort(savePort);
END; (of DoZoomWindow)
```

If the user is zooming the window to the standard state, `DoZoomWindow` calculates a new standard size and location based on the application's own considerations, the current location of the window, and the available screens. The `DoZoomWindow` procedure always places the standard state on the screen where the window is currently displayed or, if the window spans screens, on the screen containing the largest area of the window.

Listing 5-3 uses the QuickDraw routines `GetDeviceList`, `TestDeviceAttribute`, `GetNextDevice`, `SectRect`, and `GetMainDevice` to examine characteristics of the available screens as stored in `GDevice` records. Most of the code in Listing 5-3 is devoted to determining which screen should display the window in the standard state.

IMPORTANT

Never use the `bounds` field of a `PixMap` record to determine the size of the screen; instead use the value of the `gdRect` field of the `GDevice` record for the screen, as shown in Listing 5-3. ▲

After calculating the standard state, if necessary, `DoZoomWindow` calls the `ZoomWindow` procedure to redraw the window frame in the new size and location and then calls the application-defined procedure `MyResizeWindow` to redraw the window's content region. For more information on zooming and resizing windows, see the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

Setting a Device's Pixel Depth

The Monitors control panel is the user interface for changing the pixel depth, color capabilities, and positions of video devices. Since the user can control the capabilities of the video device, your application should be flexible: although it may have a preferred pixel depth, your application should do its best to accommodate less than ideal conditions.

Your application can use the `SetDepth` function to change the pixel depth of a video device, but your application should do so only with the consent of the user. If your application must have a specific pixel depth, it can display a dialog box that offers the user a choice between changing to that depth or canceling display of the image. This dialog box saves the user the trouble of going to the Monitors control panel before returning to your application. (See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for more information about creating and using dialog boxes.)

Before calling `SetDepth`, use the `HasDepth` function to determine whether the available hardware can support the pixel depth you require. The `SetDepth` function is described on page 5-34, and the `HasDepth` function is described on page 5-33.

Exceptional Cases When Working With Color Devices

If your application always specifies colors in `RGBColor` records, Color QuickDraw automatically handles the colors on both indexed and direct devices. However, if your application does not specify colors in `RGBColor` records, your application may need to create and use special-purpose `CGrafPort`, `PixMap`, and `GDevice` records with the routines described in the chapter "Offscreen Graphics Worlds."

If your application must work with `CGrafPort`, `PixMap`, and `GDevice` records in ways beyond the scope of the routines described elsewhere in this book, the following guidelines may aid you in adapting Color QuickDraw to color graphics devices.

- Don't draw directly to the screen. Create your own offscreen graphics world (as described in the chapter "Offscreen Graphics Worlds") and use the `CopyBits`, `CopyMask`, or `CopyDeepMask` routine (described in the chapter "Color QuickDraw") to transfer the image to the screen.
- Don't directly change the `fgColor` or `bkColor` fields of a `GrafPort` record and expect them to be used as the pixel values. Color QuickDraw recalculates these values for each graphics device. If you want to draw with a color with a particular *index value*, use a palette with explicit colors, as described in *Inside Macintosh: Advanced Color Imaging*. For device-independent colors, use the `RGBForeColor` and `RGBBackColor` procedures, described in the chapter "Color QuickDraw" in this book.

- Don't copy a `GDevice` record's `PixMap` record. Instead, use the `NewPixMap` function or the `CopyPixMap` procedure, and fill all the fields. (These routines are described in the chapter "Color QuickDraw.") The `NewPixMap` function returns a `PixMap` record that is cloned from the `PixMap` record pointed to by the global variable `TheGDevice`. If you don't want a copy of the main screen's `PixMap` record—for example, you want one that is a different pixel depth—then you must fill out more fields than just `pixelSize`: you must fill out the `pixelType`, `cmpCount`, and `cmpSize` fields. Set the `pmVersion` field to 0 when initializing your own `PixMap` record. For future compatibility you should also set the `packType`, `packSize`, `planeBytes`, and `pmReserved` fields to 0. Don't assume a `PixMap` record has a color table—a pixel map for a direct device doesn't need one. For compatibility, a `PixMap` record for a direct device should have a dummy handle in the `pmTable` field that points to a `ColorTable` record with a seed value equal to `cmpSize` × `cmpCount` and a `ctSize` field set to 0.
- Fill out all the fields of a new `GDevice` record. When creating an offscreen `GDevice` record by calling `NewGDevice` with the `mode` parameter set to –1, you must fill out the fields of the `GDevice` record (for instance, the `gdType` field) yourself. If you want a copy of an existing `GDevice` record, copy the `gdType` field from it. If you explicitly want an indexed device, assign the `clutType` constant to the `gdType` field.

Graphics Devices Reference

This section describes the `GDevice` record, the routines that manipulate `GDevice` records, and the `'scrn'` resource.

"Data Structures" shows the Pascal data structure for the `GDevice` record, which contains information about a video device or offscreen graphics world. "Data Structures" also shows the data structure for the `DeviceLoopFlags` data type, which defines a set of options you can specify to the `DeviceLoop` procedure.

"Routines for Graphics Devices" describes routines for creating, setting, and disposing of `GDevice` records; getting the available graphics devices; and determining device characteristics. Your application generally never needs to create, set, or dispose of `GDevice` records. However, you may find it useful for your application to get `GDevice` records to determine the capabilities of the user's screens. When zooming a window, for example, your application can use `GDevice` records to determine which screen contains the largest area of a window, and then determine the ideal window size for that screen. You may also wish to use the `DeviceLoop` procedure, described in this chapter, if you want to optimize your application's drawing for graphics devices with different capabilities. "Application-Defined Routine" describes how you can define your own drawing procedure when optimizing your application's drawing for different graphics devices.

"Resource" describes the screen (`'scrn'`) resource. System software automatically creates and uses this resource; your application never needs it. The screen resource is documented here for your general information.

Data Structures

This section shows the Pascal data structure for the `GDevice` record, which can contain information about a video device or an offscreen graphics world. This section also shows the data structure for the `DeviceLoopFlags` data type, which defines a set of options you can specify to the `DeviceLoop` procedure.

GDevice

Color QuickDraw stores state information for video devices and offscreen graphics worlds in `GDevice` records. When the system starts up, it allocates and initializes one handle to a `GDevice` record for each video device it finds. When you use the `NewGWorld` function (described in the chapter "Offscreen Graphics Worlds" in this book), Color QuickDraw automatically creates a `GDevice` record for the new offscreen graphics world. The system links these `GDevice` records in a list, called the *device list*. (You can find a handle to the first element in the device list in the global variable `DeviceList`.) By default, the `GDevice` record corresponding to the first video device found is marked as the current device; all other graphics devices in the list are initially marked as inactive.

Note
Printing graphics ports, described in the chapter "Printing Manager" in this book, do not have `GDevice` records. ◆

When the user moves a window or creates a window on another screen, and your application draws into that window, Color QuickDraw automatically makes the video device for that screen the current device. Color QuickDraw stores that information in the global variable `TheGDevice`.

`GDevice` records that correspond to video devices have drivers associated with them. These drivers can be used to change the mode of the video device from black and white to color and to change the pixel depth. The set of routines supported by a video driver is defined and described in *Designing Cards and Drivers for the Macintosh Family*, third edition. Application-created `GDevice` records usually don't require drivers.

A `GDevice` record is defined as follows:

```
TYPE GDevice =
RECORD
   gdRefNum:      Integer;       {reference number of screen }
                                 { driver}
   gdID:          Integer;       {reserved; set to 0}
   gdType:        Integer;       {device type--indexed or direct}
   gdITable:      ITabHandle;    {handle to inverse table for }
                                 { Color Manager}
   gdResPref:     Integer;       {preferred resolution}
```

```
   gdSearchProc:  SProcHndl;     {handle to list of search }
                                 { functions}
   gdCompProc:    CProcHndl;     {handle to list of complement }
                                 { functions}
   gdFlags:       Integer;       {graphics device flags}
   gdPMap:        PixMapHandle;  {handle to PixMap record for }
                                 { displayed image}
   gdRefCon:      LongInt;       {reference value}
   gdNextGD:      GDHandle;      {handle to next graphics device}
   gdRect:        Rect;          {graphics device's global bounds}
   gdMode:        LongInt;       {graphics device's current mode}
   gdCCBytes:     Integer;       {width of expanded cursor data}
   gdCCDepth:     Integer;       {depth of expanded cursor data}
   gdCCXData:     Handle;        {handle to cursor's expanded }
                                 { data}
   gdCCXMask:     Handle;        {handle to cursor's expanded }
                                 { mask}
   gdReserved:    LongInt;       {reserved for future use--must }
                                 { be 0}
END;
```

Field descriptions

`gdRefNum` The reference number of the driver for the screen associated with the video device. For most video devices, this information is set at system startup time.

`gdID` Reserved. If you create your own `GDevice` record, set this field to 0.

`gdType` The general type of graphics device. Values include

```
CONST
clutType = 0;     {CLUT device--that is, one with }
                  { colors mapped with a color }
                  { lookup table}
fixedType = 1;    {fixed colors--that is, the }
                  { color lookup table can't }
                  { be changed}
directType = 2;   {direct RGB colors}
```

These types are described in more detail in the chapter "Color Manager" in *Inside Macintosh: Advanced Color Imaging*.

`gdITable` A handle to the inverse table for color mapping; the inverse table is described in the chapter "Color Manager" in *Inside Macintosh: Advanced Color Imaging*.

`gdResPref` The preferred resolution for inverse tables.

`gdSearchProc` A handle to the list of search functions, as described in the chapter "Color Manager" in *Inside Macintosh: Advanced Color Imaging*; its value is `NIL` for the default function.

`gdCompProc` A handle to a list of complement functions, as described in the chapter "Color Manager" in *Inside Macintosh: Advanced Color Imaging*; its value is `NIL` for the default function.

`gdFlags` The `GDevice` record's attributes. To set the attribute bits in the `gdFlags` field, use the `SetDeviceAttribute` procedure (described on page 5-22)—do not set these flags directly in the `GDevice` record. The constants representing each bit are listed here.

```
CONST {flag bits for gdFlags field of GDevice record}
gdDevType      = 0;  {if bit is set to 0, graphics device is }
                     { black and white; if set to 1, }
                     { graphics device supports color}
burstDevice    = 7;  {if bit is set to 1, graphics device }
                     { supports block transfer}
ext32Device    = 8;  {if bit is set to 1, graphics device }
                     { must be used in 32-bit mode}
ramInit        = 10; {if bit is set to 1, graphics device has }
                     { been initialized from RAM}
mainScreen     = 11; {if bit is set to 1, graphics device is }
                     { the main screen}
allInit        = 12; {if bit is set to 1, all graphics devices }
                     { were initialized from 'scrn' resource}
screenDevice   = 13; {if bit is set to 1, graphics device is }
                     { a screen}
noDriver       = 14; {if bit is set to 1, GDevice }
                     { record has no driver}
screenActive   = 15; {if bit is set to 1, graphics device is }
                     { active}
```

`gdPMap` A handle to a `PixMap` record giving the dimension of the image buffer, along with the characteristics of the graphics device (resolution, storage format, color depth, and color table). `PixMap` records are described in the chapter "Color QuickDraw" in this book. For `GDevice` records, the high bit of the global variable `TheGDevice^^.gdPMap^^.pmTable^^.ctFlags` is always set.

`gdRefCon` A value used by system software to pass device-related parameters. Since a graphics device is shared, you shouldn't store data here.

`gdNextGD` A handle to the next graphics device in the device list. If this is the last graphics device in the device list, the field contains 0.

`gdRect`	The boundary rectangle of the graphics device represented by the `GDevice` record. The main screen has the upper-left corner of the rectangle set to (0,0). All other graphics devices are relative to this point.
`gdMode`	The current setting for the graphics device mode. This value is passed to the video driver to set its pixel depth and to specify color or black and white; applications don't need this information. See *Designing Cards and Drivers for the Macintosh Family,* third edition, for more information about the modes specified in this field.
`gdCCBytes`	The `rowBytes` value of the expanded cursor. Your application should not change this field. Cursors are described in the chapter "Cursor Utilities."
`gdCCDepth`	The depth of the expanded cursor. Your application should not change this field.
`gdCCXData`	A handle to the cursor's expanded data. Your application should not change this field.
`gdCCXMask`	A handle to the cursor's expanded mask. Your application should not change this field.
`gdReserved`	Reserved for future expansion; it must be set to 0 for future compatibility.

Your application should never need to directly change the fields of a `GDevice` record. If you find it absolutely necessary for your application to so, immediately use the `GDeviceChanged` procedure to notify Color QuickDraw that your application has changed the `GDevice` record. The `GDeviceChanged` procedure is described in the chapter "Color QuickDraw" in this book.

DeviceLoopFlags

When you use the `DeviceLoop` procedure (described on page 5-29), you can change its default behavior by using the `flags` parameter to specify one or more members of the set of flags defined by the `DeviceLoopFlags` data type. These flags are described here; if you want to use the default behavior of `DeviceLoop`, pass in the `flags` parameter 0 in your C code or an empty set ([ ]) in your Pascal code.

```
TYPE DeviceLoopFlags =
SET OF                {for flags parameter of DeviceLoop}
   (singleDevices,    {DeviceLoop doesn't group similar graphics }
                      { devices when calling drawing procedure}
   dontMatchSeeds,    {DeviceLoop doesn't consider ctSeed fields }
                      { of ColorTable records for graphics }
                      { devices when comparing them}
   allDevices);       {DeviceLoop ignores value of drawingRgn }
                      { parameter--instead, it calls drawing }
                      { procedure for every screen}
```

Field descriptions

`singleDevices` If this flag is not set, `DeviceLoop` calls your drawing procedure only once for each set of similar graphics devices, and the first one found is passed as the target device. (It is assumed to be representative of all the similar graphics devices.) If you set the `singleDevices` flag, then `DeviceLoop` does not group similar graphics devices—that is, those having identical pixel depths, black-and-white or color settings, and matching color table seeds—when it calls your drawing procedure.

`dontMatchSeeds`

If you set the `dontMatchSeeds` flag, then `DeviceLoop` doesn't consider color table seeds when comparing graphics devices for similarity; `DeviceLoop` ignores this flag if you set the `singleDevices` flag. Used primarily by the Palette Manager, the `ctSeed` field of a `ColorTable` record is described in the chapter "Color QuickDraw" in this book.

`allDevices` If you set the `allDevices` flag, `DeviceLoop` ignores the `drawingRgn` parameter and calls your drawing procedure for every graphics device. The value of current graphics port's `visRgn` field is not affected when you set this flag.

Routines for Graphics Devices

This section describes routines for creating, setting, and disposing of `GDevice` records; for getting the available video devices and offscreen graphics worlds; and for determining the characteristics of video devices and offscreen graphics worlds. Generally, your application won't need to use the routines for creating, setting, and disposing of `GDevice` records, because Color QuickDraw calls them automatically as appropriate. However, you may wish to use the other routines described in this section, particularly if you want to optimize your application's drawing for screens with different capabilities.

Creating, Setting, and Disposing of GDevice Records

Color QuickDraw uses `GDevice` records to maintain information about video devices and offscreen graphics worlds. A `GDevice` record must be allocated with the `NewGDevice` function and initialized with the `InitGDevice` procedure. Normally, your application does not call these routines directly. When the system starts up, it allocates and initializes one handle to a `GDevice` record for each video device it finds. When you use the `NewGWorld` function (described in the chapter "Offscreen Graphics Worlds" in this book), Color QuickDraw automatically creates a `GDevice` record for the new offscreen graphics world.

Whenever QuickDraw routines are used to draw into a graphics port on a video device, Color QuickDraw uses the `SetGDevice` procedure to make the video device for that screen the current device. Your application won't generally need to use this procedure, because when your application draws into a window on one or more screens, Color QuickDraw automatically switches `GDevice` records as appropriate; and when your application needs to draw into an offscreen graphics world, it can use the `SetGWorld` procedure to set the graphics port as well as the `GDevice` record for the offscreen environment. However, if your application uses the `SetPort` procedure (described in the chapter "Basic QuickDraw" in this book) instead of the `SetGWorld` procedure to set the graphics port to or from an offscreen graphics world, then your application must use `SetGDevice` in conjunction with `SetPort`.

You use the `SetDeviceAttribute` procedure to set attribute bits in a `GDevice` record.

When Color QuickDraw no longer needs a `GDevice` record, it uses the `DisposeGDevice` procedure to dispose of it. As with the other routines described in this section, your application typically does not need to use `DisposeGDevice`.

NewGDevice

You can use the `NewGDevice` function to create a new `GDevice` record, although you generally don't need to, because Color QuickDraw uses this function to create `GDevice` records for your application automatically.

```
FUNCTION NewGDevice (refNum: Integer; mode: LongInt): GDHandle;
```

`refNum`	Reference number of the graphics device for which you are creating a `GDevice` record. For most video devices, this information is set at system startup.
`mode`	The device configuration mode. Used by the screen driver, this value sets the pixel depth and specifies color or black and white.

DESCRIPTION

For the graphics device whose driver is specified in the `refNum` parameter and whose mode is specified in the `mode` parameter, the `NewGDevice` function allocates a new `GDevice` record and all of its handles, and then calls the `InitGDevice` procedure to initialize the record. As its function result, `NewGDevice` returns a handle to the new `GDevice` record. If the request is unsuccessful, `NewGDevice` returns `NIL`.

The `NewGDevice` function allocates the new `GDevice` record and all of its handles in the system heap, and the `NewGDevice` function sets all attributes in the `gdFlags` field of the `GDevice` record to `FALSE`. If your application creates a `GDevice` record, it can use the `SetDeviceAttribute` procedure, described on page 5-22, to change the flag bits in the `gdFlags` field of the `GDevice` record to `TRUE`. Your application should never directly change the `gdFlags` field of the `GDevice` record; instead, your application should use only the `SetDeviceAttribute` procedure.

If your application creates a `GDevice` record without a driver, it should set the `mode` parameter to –1. In this case, `InitGDevice` cannot initialize the `GDevice` record, so your application must perform all initialization of the record. A `GDevice` record's default mode is defined as 128; this is assumed to be a black-and-white mode. If you specify a value other than 128 in the `mode` parameter, the record's `gdDevType` bit in the `gdFlags` field of the `GDevice` record is set to `TRUE` to indicate that the graphics device is capable of displaying color.

The `NewGDevice` function doesn't automatically insert the `GDevice` record into the device list. In general, your application shouldn't create `GDevice` records, and if it ever does, it should never add them to the device list.

SPECIAL CONSIDERATIONS

If your program uses `NewGDevice` to create a graphics device without a driver, `InitGDevice` does nothing; instead, your application must initialize all fields of the `GDevice` record. After your application initializes the color table for the `GDevice` record, your application should call the Color Manager procedure `MakeITable` to build the inverse table for the graphics device.

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

SEE ALSO

The `GDevice` record is described on page 5-15. See *Designing Cards and Drivers for the Macintosh Family,* third edition, for more information about the device modes that you can specify in the `mode` parameter. The Color Manager is described in *Inside Macintosh: Advanced Color Imaging*.

InitGDevice

The `NewGDevice` function uses the `InitGDevice` procedure to initialize a `GDevice` record.

```
PROCEDURE InitGDevice (gdRefNum: Integer; mode: LongInt;
                       gdh: GDHandle);
```

`gdRefNum`	Reference number of the graphics device. System software sets this number at system startup time for most graphics devices.
`mode`	The device configuration mode. Used by the screen driver, this value sets the pixel depth and specifies color or black and white.
`gdh`	The handle, returned by the `NewGDevice` function, to the `GDevice` record to be initialized.

DESCRIPTION

The `InitGDevice` procedure initializes the `GDevice` record specified in the `gdh` parameter. The `InitGDevice` procedure sets the graphics device whose driver has the reference number specified in the `gdRefNum` parameter to the mode specified in the `mode` parameter. The `InitGDevice` procedure then fills out the `GDevice` record, previously created with the `NewGDevice` function, to contain all information describing that mode.

The `mode` parameter determines the configuration of the device; possible modes for a device can be determined by interrogating the video device's ROM through Slot Manager routines. The information describing the device's mode is primarily contained in the video device's ROM. If the video device has a fixed color table, then that table is read directly from the ROM. If the video device has a variable color table, then `InitGDevice` uses the default color table defined in a `'clut'` resource, contained in the System file, that has a resource ID equal to the video device's pixel depth.

In general, your application should never need to call `InitGDevice`. All video devices are initialized at start time, and users change modes through the Monitors control panel.

SPECIAL CONSIDERATIONS

If your program uses `NewGDevice` to create a graphics device without a driver, `InitGDevice` does nothing; instead, your application must initialize all fields of the `GDevice` record. After your application initializes the color table for the `GDevice` record, your application should call the Color Manager procedure `MakeITable` to build the inverse table for the graphics device.

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

SEE ALSO

The `GDevice` record is described on page 5-15. See *Designing Cards and Drivers for the Macintosh Family,* third edition, for more information about the device modes that you can specify in the `mode` parameter. The `MakeITable` procedure is described in the chapter "Color Manager" in *Inside Macintosh: Advanced Color Imaging*.

SetDeviceAttribute

To set the attribute bits of a `GDevice` record, use the `SetDeviceAttribute` procedure.

```
PROCEDURE SetDeviceAttribute (gdh: GDHandle; attribute: Integer;
                              value: Boolean);
```

`gdh` A handle to a `GDevice` record.

`attribute` One of the following constants, which represent bits in the `gdFlags` field of a `GDevice` record:

```
CONST {flag bits for gdFlags field of GDevice record}
 gdDevType      = 0;  {if bit is set to 0, graphics }
                      { device is black and white; }
                      { if set to 1, device supports }
                      { color}
 burstDevice    = 7;  {if bit is set to 1, device }
                      { supports block transfer}
 ext32Device    = 8;  {if bit is set to 1, device }
                      { must be used in 32-bit mode}
 ramInit        = 10; {if bit is set to 1, device has }
                      { been initialized from RAM}
 mainScreen     = 11; {if bit is set to 1, device is }
                      { the main screen}
 allInit        = 12; {if bit is set to 1, all }
                      { devices were initialized from }
                      { 'scrn' resource}
 screenDevice   = 13; {if bit is set to 1, device is }
                      { a screen}
 noDriver       = 14; {if bit is set to 1, GDevice }
                      { record has no driver}
 screenActive   = 15; {if bit is set to 1, device is }
                      { active}
```

`value` A value of either 0 or 1 for the flag bit specified in the `attribute` parameter.

DESCRIPTION

For the graphics device specified in the `gdh` parameter, the `SetDeviceAttribute` procedure sets the flag bit specified in the `attribute` parameter to the value specified in the `value` parameter.

SPECIAL CONSIDERATIONS

Your application should never directly change the `gdFlags` field of the `GDevice` record; instead, your application should use only the `SetDeviceAttribute` procedure.

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

SetGDevice

Your application can use the `SetGDevice` procedure to set a `GDevice` record as the current device.

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

`gdh` A handle to a `GDevice` record.

DESCRIPTION

The `SetGDevice` procedure sets the specified `GDevice` record as the current device. Your application won't generally need to use this procedure, because when your application draws into a window on one or more screens, Color QuickDraw automatically switches `GDevice` records as appropriate; and when your application needs to draw into an offscreen graphics world, it can use the `SetGWorld` procedure to set the graphics port as well as the `GDevice` record for the offscreen environment. However, if your application uses the `SetPort` procedure (described in the chapter "Basic QuickDraw" in this book) instead of the `SetGWorld` procedure to set the graphics port to or from an offscreen graphics world, then your application must use `SetGDevice` in conjunction with `SetPort`.

A handle to the currently active device is kept in the global variable `TheGDevice`.

SPECIAL CONSIDERATIONS

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

SEE ALSO

See the chapter "Offscreen Graphics Worlds" in this book for information about the `SetGWorld` procedure and about drawing into offscreen graphics worlds.

DisposeGDevice

Although your application generally should never need to use this routine, the `DisposeGDevice` procedure disposes of a `GDevice` record, releases the space allocated for it, and disposes of all the data structures allocated for it. The `DisposeGDevice` procedure is also available as the `DisposGDevice` procedure.

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

`gdh` A handle to the `GDevice` record.

DESCRIPTION

The `DisposeGDevice` procedure disposes of a `GDevice` record, releases the space allocated for it, and disposes of all the data structures allocated for it. Color QuickDraw calls this procedure when appropriate.

SEE ALSO

When your application uses the `DisposeGWorld` procedure to dispose of an offscreen graphics world, `DisposeGDevice` disposes of its `GDevice` record. See the chapter "Offscreen Graphics Worlds" in this book for a description of `DisposeGWorld`.

Getting the Available Graphics Devices

To gain access to the `GDevice` record for a video device—for example, to determine the size and pixel depth of its attached screen—your application needs to get a handle to that record.

Your application can use the `GetDeviceList` function to obtain a handle to the first `GDevice` record in the device list, the `GetGDevice` function to obtain a handle to the `GDevice` record for the current device, the `GetMainDevice` function to obtain a handle to the `GDevice` record for the main screen, and the `GetMaxDevice` function to obtain a handle to the `GDevice` record for the video device with the greatest pixel depth.

All existing `GDevice` records are linked together in the device list. After using one of these functions to obtain a handle to one of the `GDevice` records in the device list, your application can use the `GetNextDevice` function to obtain a handle to the next `GDevice` record in the list.

Two related functions, `GetGWorld` and `GetGWorldDevice`, also allow you to obtain handles to `GDevice` records. To get the `GDevice` record for the current device, you can use the `GetGWorld` function. To get a handle to the `GDevice` record for a particular offscreen graphics world, you can use the `GetGWorldDevice` function. These two functions are described in the next chapter, "Offscreen Graphics Worlds."

GetGDevice

To obtain a handle to the `GDevice` record for the current device, use the `GetGDevice` function.

```
FUNCTION GetGDevice: GDHandle;
```

DESCRIPTION

The `GetGDevice` function returns a handle to the `GDevice` record for the current device. (At any given time, exactly one video device is the current device—that is, the one on which drawing is actually taking place.)

Color QuickDraw stores a handle to the current device in the global variable `TheGDevice`.

SPECIAL CONSIDERATIONS

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

GetDeviceList

To obtain a handle to the first `GDevice` record in the device list, use the `GetDeviceList` function.

```
FUNCTION GetDeviceList: GDHandle;
```

DESCRIPTION

The `GetDeviceList` function returns a handle to the first `GDevice` record in the global variable `DeviceList`.

SPECIAL CONSIDERATIONS

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

SEE ALSO

Listing 5-3 on page 5-10 illustrates the use of this function.

GetMainDevice

To obtain a handle to the `GDevice` record for the main screen, use the `GetMainDevice` function.

```
FUNCTION GetMainDevice: GDHandle;
```

DESCRIPTION

The `GetMainDevice` function returns a handle to the `GDevice` record that corresponds to the main screen—that is, the one containing the menu bar.

A handle to the main device is kept in the global variable `MainDevice`.

SPECIAL CONSIDERATIONS

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

SEE ALSO

Listing 5-3 on page 5-10 illustrates the use of this function.

GetMaxDevice

To obtain a handle to the `GDevice` record for the video device with the greatest pixel depth, use the `GetMaxDevice` function.

```
FUNCTION GetMaxDevice (globalRect: Rect): GDHandle;
```

`globalRect`
A rectangle, in global coordinates, that intersects the graphics devices that you are searching to find the one with the greatest pixel depth.

DESCRIPTION

As its function result, `GetMaxDevice` returns a handle to the `GDevice` record for the video device that has the greatest pixel depth among all graphics devices that intersect the rectangle you specify in the `globalRect` parameter.

SPECIAL CONSIDERATIONS

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

GetNextDevice

After using the `GetDeviceList` function to obtain a handle to the first `GDevice` record in the device list, `GetGDevice` to obtain a handle to the `GDevice` record for the current device, `GetMainDevice` to obtain a handle to the `GDevice` record for the main screen, or `GetMaxDevice` to obtain a handle to the `GDevice` record for the video device with the greatest pixel depth, you can use the `GetNextDevice` function to obtain a handle to the next `GDevice` record in the list.

```
FUNCTION GetNextDevice (curDevice: GDHandle): GDHandle;
```

`curDevice` A handle to the `GDevice` record at which you want the search to begin.

DESCRIPTION

The `GetNextDevice` function returns a handle to the next `GDevice` record in the device list. If there are no more `GDevice` records in the list, it returns `NIL`.

SPECIAL CONSIDERATIONS

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

SEE ALSO

Listing 5-3 on page 5-10 illustrates the use of this function.

Determining the Characteristics of a Video Device

For drawing images that are optimized for every screen they cross, your application can use the `DeviceLoop` procedure. The `DeviceLoop` procedure searches for graphics devices that intersect your window's drawing region, and it calls your drawing procedure for each different video device it finds. The `DeviceLoop` procedure provides your drawing procedure with information about the current device's pixel depth and other attributes.

To determine whether the flag bit for an attribute has been set in the `gdFlags` field of a `GDevice` record, your application can use the `TestDeviceAttribute` function.

To determine whether a video device supports a specific pixel depth, your application can also use the `HasDepth` function, described on page 5-33. To change the pixel depth of a video device, your application can use the `SetDepth` function, described on page 5-34.

If you need to determine the resolution of the main device, you can use the `ScreenRes` procedure.

DeviceLoop

For drawing images that are optimized for every screen they cross, use the `DeviceLoop` procedure.

```
PROCEDURE DeviceLoop (drawingRgn: RgnHandle;
                      drawingProc: DeviceLoopDrawingProcPtr;
                      userData: LongInt; flags: DeviceLoopFlags);
```

`drawingRgn`
A handle to the region in which you will draw; this drawing region uses coordinates that are local to its graphics port.

`drawingProc`
A pointer to your own drawing procedure.

`userData` Any additional data that you wish to supply to your drawing procedure.

`flags` One or more members of the set of flags defined by the `DeviceLoopFlags` data type:

```
TYPE
   DeviceLoopFlags = SET OF
   (singleDevices,dontMatchSeeds,allDevices);
```

These flags are described in the following text; if you want to use the default behavior of `DeviceLoop`, specify an empty set ([]) in this parameter.

DESCRIPTION

The `DeviceLoop` procedure searches for graphics devices that intersect your window's drawing region, and it calls your drawing procedure for each video device it finds. In the `drawingRgn` parameter, supply a handle to the region in which you wish to draw; in the `drawingProc` parameter, supply a pointer to your drawing procedure. In the `flags` parameter, you can specify members of the set of these flags defined by the `DeviceLoopFlags` data type:

`singleDevices` If this flag is not set, `DeviceLoop` calls your drawing procedure only once for each set of similar graphics devices, and the first one found is passed as the target device. (It is assumed to be representative of all the similar graphics devices.) If you set the `singleDevices` flag, then `DeviceLoop` does not group similar graphics devices—that is, those having identical pixel depths, black-and-white or color settings, and matching color table seeds—when it calls your drawing procedure.

`dontMatchSeeds` If you set the `dontMatchSeeds` flag, then `DeviceLoop` doesn't consider the `ctSeed` field of `ColorTable` records for graphics devices when comparing them; `DeviceLoop` ignores this flag if you set the `singleDevices` flag.

`allDevices` If you set the `allDevices` flag, `DeviceLoop` ignores the `drawingRgn` parameter and calls your drawing procedure for every device. The value of current graphics port's `visRgn` field is not affected when you set this flag.

For each dissimilar video device that intersects this region, `DeviceLoop` calls your drawing procedure. For example, after a call to the Event Manager procedure `BeginUpdate`, the region you specify in the `drawingRgn` parameter can be the same as the visible region for the active window. Because `DeviceLoop` provides your drawing procedure with the pixel depth and other attributes of each video device, your drawing procedure can optimize its drawing for each video device—for example, by using the `HiliteColor` procedure to set magenta as the highlight color on a color video device.

SPECIAL CONSIDERATIONS

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

SEE ALSO

Listing 5-1 on page 5-8 illustrates the use of `DeviceLoop`. See page 5-35 for a description of the drawing procedure you must provide for the `drawingProc` parameter. Offscreen graphics worlds are described in the next chapter. The `HiliteColor` procedure is described in the chapter "Color QuickDraw" in this book.

TestDeviceAttribute

To determine whether the flag bit for an attribute has been set in the `gdFlags` field of a `GDevice` record, use the `TestDeviceAttribute` function.

```
FUNCTION TestDeviceAttribute (gdh: GDHandle;
                              attribute: Integer): Boolean;
```

`gdh` A handle to a `GDevice` record.

`attribute` One of the following constants, which represent bits in the `gdFlags` field of a `GDevice` record:

```
CONST {flag bits for gdFlags field of GDevice record}
 gdDevType      = 0;  {if bit is set to 0, graphics }
                      { device is black and white; }
                      { if set to 1, device supports }
                      { color}
 burstDevice    = 7;  {if bit is set to 1, device }
                      { supports block transfer}
 ext32Device    = 8;  {if bit is set to 1, device }
                      { must be used in 32-bit mode}
 ramInit        = 10; {if bit is set to 1, device has }
                      { been initialized from RAM}
 mainScreen     = 11; {if bit is set to 1, device is }
                      { the main screen}
 allInit        = 12; {if bit is set to 1, all }
                      { devices were initialized from }
                      { 'scrn' resource}
 screenDevice   = 13; {if bit is set to 1, device is }
                      { a screen}
 noDriver       = 14; {if bit is set to 1, GDevice }
                      { record has no driver}
 screenActive   = 15; {if bit is set to 1, device is }
                      { active}
```

DESCRIPTION

The `TestDeviceAttribute` function tests a single graphics device attribute to see if its bit is set to 1 and, if so, returns `TRUE`. Otherwise, `TestDeviceAttribute` returns `FALSE`.

SPECIAL CONSIDERATIONS

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

SEE ALSO

Listing 5-3 on page 5-10 illustrates the use of `TestDeviceAttribute`. Your application can use the `SetDeviceAttribute` procedure, described on page 5-22, to change any of the flags tested by the `TestDeviceAttribute` function.

ScreenRes

If you need to determine the resolution of the main device, you can use the `ScreenRes` procedure.

```
PROCEDURE ScreenRes (VAR scrnHRes,scrnVRes: Integer);
```

DESCRIPTION

In the `scrnHRes` parameter, the `ScreenRes` procedure returns the number of horizontal pixels per inch displayed by the current device. In the `scrnVRes` parameter, it returns the number of vertical pixels per inch.

To determine the resolutions of all available graphics devices, you should examine their `GDevice` records (described on page 5-15). The horizontal and vertical resolutions for a graphics device are stored in the `hRes` and `vRes` fields, respectively, of the `PixMap` record for the device's `GDevice` record.

SPECIAL CONSIDERATIONS

Currently, QuickDraw and the Printing Manager always assume a screen resolution of 72 dpi.

Do not use the actual screen resolution as a scaling factor when drawing into a printing graphics port; instead, always use 72 dpi as the scaling factor. See the chapter "Printing Manager" in this book for more information about the Printing Manager and drawing into a printing graphics port.

ASSEMBLY-LANGUAGE INFORMATION

The horizontal resolution, in pixels per inch, is stored in the global variable `ScrHRes`, and the vertical resolution is stored in the global variable `ScrVRes`.

Changing the Pixel Depth for a Video Device

The Monitors control panel is the user interface for changing the pixel depth, color capabilities, and positions of video devices. Since the user can control the capabilities of the video device, your application should be flexible: although it may have a preferred pixel depth, your application should do its best to accommodate less than ideal conditions.

If it is absolutely necessary for your application to draw on a video device of a specific pixel depth, your application can use the `SetDepth` function to change its pixel depth. Before calling `SetDepth`, use the `HasDepth` function to determine whether the available hardware can support the pixel depth you require.

HasDepth

To determine whether a video device supports a specific pixel depth, you can use the `HasDepth` function.

```
FUNCTION HasDepth (aDevice: GDHandle; depth: Integer;
                   whichFlags: Integer; flags: Integer): Integer;
```

`aDevice` A handle to the `GDevice` record of the video device.

`depth` The pixel depth for which you're testing.

`whichFlags`
The `gdDevType` constant, which represents a bit in the `gdFlags` field of the `GDevice` record. (If this bit is set to 0 in the `GDevice` record, the video device is black and white; if the bit is set to 1, the device supports color.)

`flags` The value 0 or 1. If you pass 0 in this parameter, the `HasDepth` function tests whether the video device is black and white; if you pass 1 in this parameter, `HasDepth` tests whether the video device supports color.

DESCRIPTION

The `HasDepth` function checks whether the video device you specify in the `aDevice` parameter supports the pixel depth you specify in the `depth` parameter, and whether the device is black and white or color, whichever you specify in the `flags` parameter.

The `HasDepth` function returns 0 if the device does not support the depth you specify in the `depth` parameter or the display mode you specify in the `flags` parameter.

Any other value indicates that the device supports the specified depth and display mode. The function result contains the mode ID that QuickDraw passes to the video driver to set its pixel depth and to specify color or black and white. You can pass this mode ID in the `depth` parameter for the `SetDepth` function (described next) to set the graphics device to the pixel depth and display mode for which you tested.

SPECIAL CONSIDERATIONS

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

SEE ALSO

See *Designing Cards and Drivers for the Macintosh Family,* third edition, for more information about the device modes returned as a function result for `HasDepth`.

SetDepth

To change the pixel depth of a video device, use the `SetDepth` function.

```
FUNCTION SetDepth (aDevice: GDHandle; depth: Integer;
                   whichFlags: Integer; flags: Integer): OSErr;
```

`aDevice` A handle to the `GDevice` record of the video device whose pixel depth you wish to change.

`depth` The mode ID returned by the `HasDepth` function (described in the previous section) indicating that the video device supports the desired pixel depth. Alternatively, you can pass the desired pixel depth directly in this parameter, although you should use the `HasDepth` function to ensure that the device supports this depth.

`whichFlags` The `gdDevType` constant, which represents a bit in the `gdFlags` field of the `GDevice` record. (If this bit is set to 0 in the `GDevice` record, the video device is black and white; if the bit is set to 1, the device supports color.)

`flags` The value 0 or 1. If you pass 0 in this parameter, the `SetDepth` function changes the video device to black and white; if you pass 1 in this parameter, `SetDepth` changes the video device to color.

DESCRIPTION

The `SetDepth` function sets the video device you specify in the `aDevice` parameter to the pixel depth you specify in the `depth` parameter, and it sets the device to either black and white or color as you specify in the `flags` parameter. You should use the `HasDepth` function to ensure that the video device supports the values you specify to `SetDepth`. The `SetDepth` returns zero if successful, or it returns a nonzero value if it cannot impose the desired depth and display mode on the requested device.

The `SetDepth` function does not change the `'scrn'` resource; when the system is restarted, the original depth for this device is restored.

SPECIAL CONSIDERATIONS

Your application should use `SetDepth` only if your application can run on devices of a particular pixel depth and is unable to adapt to any other depth. Your application should display a dialog box that offers the user a choice between changing to that depth or canceling display of the image before your application uses `SetDepth`. Such a dialog box saves the user the trouble of going to the Monitors control panel before returning to your application.

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

SEE ALSO

See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information about creating and using dialog boxes.

Application-Defined Routine

Your application can use the `DeviceLoop` procedure (described on page 5-29) before drawing images that are optimized for every screen they cross. The `DeviceLoop` procedure searches for video devices that intersect your drawing region, and it calls a drawing procedure that you define for every different video device it finds.

For each video device that intersects a drawing region that you define (generally, the update region of a window), `DeviceLoop` calls your drawing procedure. Because `DeviceLoop` provides your drawing procedure with the pixel depth and other attributes of the current device, your drawing procedure can optimize its drawing for whatever type of graphics device is the current device. When highlighting, for example, your application might invert black and white when drawing onto a 1-bit video device but use magenta as the highlight color when drawing onto a color video device. In this case, even were your window to span both a black-and-white and a color screen, the user sees the selection inverted on the black-and-white screen, while magenta would be used to highlight the selection on the color screen.

You must provide a pointer to your drawing procedure in the `drawingProc` parameter for `DeviceLoop`.

MyDrawingProc

Here's how to declare a drawing procedure to supply to the `DeviceLoop` procedure if you were to name the procedure `MyDrawingProc`:

```
PROCEDURE MyDrawingProc (depth: Integer; deviceFlags: Integer;
                         targetDevice: GDHandle;
                         userData: LongInt);
```

`depth` The pixel depth of the graphics device.

`deviceFlags`
Any of the following constants, which represent bits that are set to 1 in the `gdFlags` field of the `GDevice` record (described on page 5-15) for the current device:

```
CONST {flag bits for gdFlags field of GDevice record}
 gdDevType      = 0;  {if bit is set to 1, graphics }
                      { device supports color}
 burstDevice    = 7;  {if bit is set to 1, device }
                      { supports block transfer}
 ext32Device    = 8;  {if bit is set to 1, device }
                      { must be used in 32-bit mode}
 ramInit        = 10; {if bit is set to 1, device has }
                      { been initialized from RAM}
 mainScreen     = 11; {if bit is set to 1, device is }
                      { the main screen}
 allInit        = 12; {if bit is set to 1, all }
                      { devices were initialized from }
                      { 'scrn' resource}
 screenDevice   = 13; {if bit is set to 1, device is }
                      { a screen}
 noDriver       = 14; {if bit is set to 1, GDevice }
                      { record has no driver}
 screenActive   = 15; {if bit is set to 1, device is }
                      { active}
```

`targetDevice`
A handle to the `GDevice` record (described on page 5-15) for the current device.

`userData` A value that your application supplies to the `DeviceLoop` procedure, which in turn passes the value to your drawing procedure for whatever purpose you deem useful.

DESCRIPTION

Your drawing procedure should analyze the pixel depth passed in the `depth` parameter and the values passed in the `deviceFlags` parameter, and then draw in a manner that is optimized for the current device.

SEE ALSO

Listing 5-2 on page 5-9 illustrates a simple drawing procedure called by `DeviceLoop`.

Resource

The user can use the Monitors control panel to set the desired pixel depth of each screen; whether it displays color, grayscale, or black and white; and the position of each screen relative to the main screen. The Monitors control panel stores all configuration information for a multiscreen system in the System file in a resource of type `'scrn'` that has a resource ID of 0. Your application should never create this resource, and should never alter or examine it.

When the `InitGraf` procedure (described in the chapter "Basic QuickDraw" in this book) initializes Color QuickDraw, it checks the System file for the `'scrn'` resource. If the `'scrn'` resource is found and it matches the hardware, `InitGraf` organizes the video devices according to the contents of this resource; if not, then Color QuickDraw uses only the video device for the startup screen.

The Screen Resource

The `'scrn'` resource consists of an array of data structures that are analogous to `GDevice` records. Each data structure in this array contains information about a different video device. Because your application shouldn't create or alter the `'scrn'` resource, its structure is not described here.

Summary of Graphics Devices

Pascal Summary

Constants

```
CONST
   {flag bits for gdType field of GDevice record}
   clutType = 0;     {CLUT device--that is, one with colors mapped with a }
                     { color lookup table}
   fixedType = 1;    {fixed colors--that is, the color lookup table }
                     { can't be changed}
   directType = 2;   {direct RGB colors}

   {flag bits for gdFlags field of GDevice record}
   gdDevType      = 0;  {if bit is set to 0, graphics device is black }
                        { and white; if bit is set to 1, graphics device }
                        { supports color}
   burstDevice    = 7;  {if bit is set to 1, graphics device supports block }
                        { transfer}
   ext32Device    = 8;  {if bit is set to 1, graphics device must be used }
                        { in 32-bit mode}
   ramInit        = 10; {if bit is set to 1, graphics device has been }
                        { initialized from RAM}
   mainScreen     = 11; {if bit is set to 1, graphics device is the main }
                        { screen}
   allInit        = 12; {if bit is set to 1, all graphics devices were }
                        { initialized from 'scrn' resource}
   screenDevice   = 13; {if bit is set to 1, graphics device is a screen}
   noDriver       = 14; {if bit is set to 1, GDevice record has no driver}
   screenActive   = 15; {if bit is set to 1, graphics device is current }
                        { device}
```

Data Types

```
TYPE
GDHandle    = ^GDPtr;
GDPtr       = ^GDevice;
GDevice     =
RECORD
   gdRefNum:      Integer;       {reference number of screen driver}
   gdID:          Integer;       {reserved; set to 0}
   gdType:        Integer;       {type of device--indexed or direct}
   gdITable:      ITabHandle;    {handle to inverse table for Color Manager}
   gdResPref:     Integer;       {preferred resolution}
   gdSearchProc:  SProcHndl;     {handle to list of search functions}
   gdCompProc:    CProcHndl;     {handle to list of complement functions}
   gdFlags:       Integer;       {graphics device flags}
   gdPMap:        PixMapHandle;  {handle to PixMap record for displayed }
                                 { image}
   gdRefCon:      LongInt;       {reference value}
   gdNextGD:      GDHandle;      {handle to next graphics device}
   gdRect:        Rect;          {graphics device's boundary in global }
                                 { coordinates}
   gdMode:        LongInt;       {graphics device's current mode}
   gdCCBytes:     Integer;       {width of expanded cursor data}
   gdCCDepth:     Integer;       {depth of expanded cursor data}
   gdCCXData:     Handle;        {handle to cursor's expanded data}
   gdCCXMask:     Handle;        {handle to cursor's expanded mask}
   gdReserved:    LongInt;       {reserved for future use; must be 0}
END;

QDErr = Integer;

DeviceLoopDrawingProcPtr = ProcPtr;

DeviceLoopFlags = SET OF   {for flags parameter of DeviceLoop}
   (singleDevices,         {DeviceLoop doesn't group similar graphics }
                           { devices when calling drawing procedure}
   dontMatchSeeds,         {DeviceLoop doesn't consider ctSeed fields }
                           { of ColorTable records for graphics devices }
                           { when comparing them}
   allDevices);            {DeviceLoop ignores value of drawingRgn }
                           { parameter--instead, it calls drawing procedure }
                           { for every screen}
```

Routines for Graphics Devices

Creating, Setting, and Disposing of GDevice Records

```
{ DisposeGDevice is also spelled as DisposGDevice }
FUNCTION NewGDevice         (refNum: Integer; mode: LongInt): GDHandle;
PROCEDURE InitGDevice       (gdRefNum: Integer; mode: LongInt;
                             gdh: GDHandle);
PROCEDURE SetDeviceAttribute
                            (gdh: GDHandle; attribute: Integer;
                             value: Boolean);
PROCEDURE SetGDevice        (gdh: GDHandle);
PROCEDURE DisposeGDevice    (gdh: GDHandle);
```

Getting the Available Graphics Devices

```
FUNCTION GetGDevice         : GDHandle;
FUNCTION GetDeviceList      : GDHandle;
FUNCTION GetMainDevice      : GDHandle;
FUNCTION GetMaxDevice       (globalRect: Rect): GDHandle;
FUNCTION GetNextDevice      (curDevice: GDHandle): GDHandle;
```

Determining the Characteristics of a Video Device

```
PROCEDURE DeviceLoop        (drawingRgn: RgnHandle;
                             drawingProc: DeviceLoopDrawingProcPtr;
                             userData: LongInt; flags: DeviceLoopFlags);
FUNCTION TestDeviceAttribute
                            (gdh: GDHandle;
                             attribute: Integer): Boolean;
PROCEDURE ScreenRes         (VAR scrnHRes,scrnVRes: Integer);
```

Changing the Pixel Depth for a Video Device

```
FUNCTION HasDepth           (aDevice: GDHandle; depth: Integer;
                             whichFlags: Integer; flags: Integer): Integer;
FUNCTION SetDepth           (aDevice: GDHandle; depth: Integer;
                             whichFlags: Integer; flags: Integer): OSErr;
```

Application-Defined Routine

```
PROCEDURE MyDrawingProc     (depth: Integer; deviceFlags: Integer;
                             targetDevice: GDHandle; userData: LongInt);
```

C Summary

Constants

```
enum {
   /* flag bits for gdType field of GDevice record */
   clutType       = 0;  /* CLUT device--that is, one with colors mapped with
                           a color lookup table */
   fixedType      = 1;  /* fixed colors--that is, the color lookup table
                           can't be changed */
   directType     = 2;  /* direct RGB colors */

   /* flag bits for gdFlags field of GDevice record */
   gdDevType      = 0,  /* if bit is set to 0, graphics device is black and
                           white; if set to 1, device is color */
   burstDevice    = 7,  /* if bit is set to 1, graphics device supports block
                           transfer */
   ext32Device    = 8,  /* if bit is set to 1, graphics device must be used
                           in 32-bit mode */
   ramInit        = 10, /* if bit is set to 1, graphics device was
                           initialized from RAM */
   mainScreen     = 11, /* if bit is set to 1, graphics device is the main
                           screen */
   allInit        = 12, /* if bit is set to 1, all graphics devices were
                           initialized from 'scrn' resource */
   screenDevice   = 13, /* if bit is set to 1, graphics device is a screen
                           device */
   noDriver       = 14, /* if bit is set to 1, GDevice record has
                           no driver */
   screenActive   = 15, /* if bit is set to 1, graphics device is current
                           device */
};
```

Data Types

```
struct GDevice {
   short          gdRefNum;      /* reference number of screen driver */
   short          gdID;          /* reserved; set to 0 */
   short          gdType;        /* type of device--indexed or direct */
   ITabHandle     gdITable;      /* handle to inverse 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 PixMap record for displayed
                                    image */
   long           gdRefCon;      /* reference value */
   GDHandle       gdNextGD;      /* handle to next graphics device */
   Rect           gdRect;        /* graphics device's boundary in global
                                    coordinates */
   long           gdMode;        /* graphics 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 for future use; must be 0 */
};
typedef struct GDevice GDevice;
typedef GDevice *GDPtr, **GDHandle;

typedef short QDErr;

typedef pascal void  (*DeviceLoopDrawingProcPtr)
                     (short depth, short deviceFlags,
                      GDHandle targetDevice, long userData);

/* for flags parameter of DeviceLoop */
enum  {singleDevicesBit = 0,dontMatchSeedsBit = 1,allDevicesBit = 2};
enum  {singleDevices = 1 << singleDevicesBit,   /* DeviceLoop doesn't group
                                                   similar graphics devices
                                                   when calling drawing
                                                   procedure */
      dontMatchSeeds = 1 << dontMatchSeedsBit,  /* DeviceLoop doesn't
                                                   consider ctSeed fields of
                                                   ColorTable records for
                                                   graphics devices when
                                                   comparing them */
      allDevices = 1 << allDevicesBit};         /* DeviceLoop ignores value
                                                   of drawingRgn parameter--
                                                   instead it calls drawing
                                                   procedure for every
                                                   screen */

typedef unsigned long DeviceLoopFlags;
```

Functions for Graphics Devices

Creating, Setting, and Disposing of GDevice Records

```
/* DisposeGDevice is also spelled as DisposGDevice */

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

Getting the Available Graphics Devices

```
pascal GDHandle GetGDevice  (void);
pascal GDHandle GetDeviceList
                            (void);
pascal GDHandle GetMainDevice
                            (void);
pascal GDHandle GetMaxDevice
                            (const Rect *globalRect);
pascal GDHandle GetNextDevice
                            (GDHandle curDevice);
```

Determining the Characteristics of a Video Device

```
pascal void DeviceLoop      (RgnHandle drawingRgn,
                             DeviceLoopDrawingProcPtr drawingProc,
                             long userData, DeviceLoopFlags flags);
pascal Boolean TestDeviceAttribute
                            (GDHandle gdh, short attribute);
pascal void ScreenRes       (short *scrnHRes, short *scrnVRes);
```

Changing the Pixel Depth for a Video Device

```
pascal Integer HasDepth     (GDHandle aDevice, Integer depth,
                             Integer whichFlags, Integer flags);
pascal OSErr SetDepth       (GDHandle aDevice, Integer depth,
                             Integer whichFlags, Integer flags);
```

Application-Defined Function

```
pascal void MyDrawingProc   (Integer depth, Integer deviceFlags,
                             GDHandle targetDevice, LongInt userData);
```

Assembly-Language Summary

Data Structure

GDevice Data Structure

0	`gdRefNum`	word	`refNum` of screen driver
2	`gdID`	word	reserved; set to 0
4	`gdType`	word	general type of graphics device
6	`gdITable`	long	handle to inverse table
10	`gdResPref`	word	preferred resolution for inverse tables
12	`gdSearchProc`	long	search function pointer
16	`gdCompProc`	long	complement function pointer
20	`gdFlags`	word	graphics device flags word
22	`gdPMap`	long	handle to pixel map describing graphics device
26	`gdRefCon`	long	reference value
30	`gdNextGD`	long	handle to next `GDevice` record
34	`gdRect`	8 bytes	graphics device's bounds in global coordinates
42	`gdMode`	long	device's current mode
46	`gdCCBytes`	word	width of expanded cursor data
48	`gdCCDepth`	word	depth of expanded cursor data
50	`gdCCXData`	long	handle to cursor's expanded data
54	`gdCCXMask`	long	handle to cursor's expanded mask
58	`gdReserved`	long	reserved; must be 0

Global Variables

`DeviceList`	Handle to the first `GDevice` record in the device list.
`MainDevice`	Handle to the `GDevice` record for the main screen.
`ScrHRes`	The horizontal resolution, in pixels per inch, for the current device.
`ScrVRes`	The vertical resolution, in pixels per inch, for the current device.
`TheGDevice`	Handle to the `GDevice` record for the current device.