Basic QuickDraw

Contents

This chapter describes how to initialize basic QuickDraw and how to create and manage a **basic graphics port**—the drawing environment in which your application can create graphics and text in either black and white or eight basic colors. Many of the routines described in this chapter also operate in color graphics ports, and are noted as such. This chapter also describes the mathematical foundation of both basic QuickDraw and Color QuickDraw.

Read this chapter to learn how to set up a drawing environment for your application on all models of Macintosh computers. The chapter "Color QuickDraw" in this book describes additional data structures and routines necessary for preparing the more sophisticated color drawing environments that are supported on the more powerful Macintosh computers.

If your application ever draws to the screen, it uses basic QuickDraw—either directly, as when it draws shapes or patterns into a window, or indirectly, as when it uses another Macintosh Toolbox manager (such as the Window Manager or Menu Manager) to implement elements of the standard Macintosh user interface. If your application does not use color, or uses only a few colors, you may find that all the tools you need for preparing a graphics environment are provided by basic QuickDraw. Once you prepare a basic drawing environment as described in this chapter, you can begin drawing into it as described in the next chapter, "QuickDraw Drawing."

About Basic QuickDraw

Basic QuickDraw, designed for the earliest Macintosh models with their built-in black-and-white screens, is a collection of system software routines that your application can use to manipulate images on all Macintosh computers.

Note

All Macintosh computers support basic QuickDraw. Only those computers based on the Motorola 68000 processor, such as the Macintosh Classic and PowerBook 100 computers, provide no support for Color QuickDraw. ◆

Basic QuickDraw performs its operations in a graphics port based on a data structure of types `GrafPort`. (Color QuickDraw, described in the chapter "Color QuickDraw," can work with data structures of type `GrafPort` or `CGrafPort`, the latter offering extensive color and grayscale facilities.)

As described in the chapter "Introduction to QuickDraw," each graphics port has its own local coordinate system. All fields in a graphics port are expressed in these coordinates, and all calculations and actions that QuickDraw performs use the local coordinate system of the current graphics port. The mathematical constructs of this coordinate system are described next.

The Mathematical Foundations of QuickDraw

QuickDraw defines some mathematical constructs that are widely used in its procedures, functions, and data types: the coordinate plane, the point, the rectangle, and the region. Points are defined in terms of the coordinate plane. Points in turn are used to define a rectangle. Rectangles assign coordinates to boundaries and images, and rectangles frame graphic objects such as regions and ovals. Regions define arbitrary areas on the coordinate plane.

For example, each graphics port has its own local coordinate system on the coordinate plane; the location of the graphics pen used for drawing into a graphics port is expressed as a point; a commonly used rectangle is the **port rectangle,** which in a graphics port for a window represents the window's content area; and a commonly used region in QuickDraw is the **visible region,** which in a graphics port for a window represents the portion of the window that's actually visible on the screen—that is, the part that's not covered by other windows.

The Coordinate Plane

As described in the chapter "Introduction to QuickDraw," all information about location or movement is specified to QuickDraw in terms of coordinates on a plane. The plane is a two-dimensional grid whose coordinates range from –32768 to 32767. On a user's computer, there is one **global coordinate system** that represents all potential QuickDraw drawing space. The origin of the global coordinate system—that is, the point with a horizontal coordinate of 0 and a vertical coordinate of 0—is at the upper-left corner of the user's main screen. Each graphics port on that user's computer has its own **local coordinate system,** which is defined relative to the port rectangle of the graphics port. Typically, the upper-left corner of a port rectangle is assigned a local horizontal coordinate of 0 and a local vertical coordinate of 0, although you can use the `SetOrigin` procedure to change the coordinates of this corner.

IMPORTANT

QuickDraw stores points and rectangles in its own data structures of types `Point` and `Rect`. In these structures, the vertical coordinate (`v`) appears first, followed by the horizontal coordinate (`h`). However, in parameters to all QuickDraw routines, you specify the horizontal coordinate first and the vertical coordinate second. ▲

So that the user can select onscreen objects across this coordinate plane, QuickDraw predefines several cursors, described in the chapter "Cursor Utilities" in this book, that the user manipulates with the mouse.

Points

A point is located by the combination of a vertical coordinate and a horizontal coordinate. Points themselves are dimensionless; if a visible pixel is located at a point, the pixel hangs down and to the right of the point. You can store the coordinates of a point into a variable of type `Point`, which QuickDraw defines as a record of two integers.

```
TYPE VHSelect =   (v,h);
Point =
RECORD
   CASE Integer OF
         0: (v:   Integer:    {vertical coordinate}
             h:   Integer);   {horizontal coordinate}
         1: (vh:  ARRAY[VHSelect] OF Integer);
END;
```

The third field of this record lets you access the vertical and horizontal coordinates of a point either individually or as an array. For example, the following code fragment illustrates how to assign values to the coordinates of points:

```
VAR
   westPt, eastPt: Point;

westPt.v := 40; westPt.h := 60;
eastPt.vh[v] := 90; eastPt.vh[h] := 110;
```

"Manipulating Points in Graphics Ports" beginning on page 2-51 describes several QuickDraw routines you can use to change and calculate points.

Rectangles

Any two points can define the upper-left and lower-right corners of a rectangle. Just as points are infinitely small, the borders of the rectangle are infinitely thin.

The data type for rectangles is `Rect`, and the data structure consists of either four integers or two points:

```
TYPE Rect =
RECORD
   CASE Integer OF                {cases: four sides or two points}
      0: (top:       Integer;     {upper boundary of rectangle}
          left:      Integer;     {left boundary of rectangle}
          bottom:    Integer;     {lower boundary of rectangle}
          right:     Integer);    {right boundary of rectangle}
      1: (topLeft:   Point;       {upper-left corner of rectangle}
          botRight:  Point);      {lower-right corner of rectangle}
END;
```

You can access a variable of type `Rect` either as four boundary coordinates or as two diagonally opposite corner points. All of the following coordinates to the rectangle named `shipRect` are permissible:

```
VAR
   shipRect: Rect;

{specify rectangle with boundary coordinates}
shipRect.top := 20; shipRect.left := 20; shipRect.bottom := 70;
   shipRect.right := 70;

{specify rectangle with upper-left and bottom-right points}
shipRect.topLeft := (20,20); shipRect.botRight := (70,70);

{specify individual coordinates for rectangle's upper-left }
{ and bottom-right points}
shipRect.topLeft.v := 20; shipRect.topLeft.h :=20;
   shipRect.botRight.v := 70; shipRect.botRight.h :70;

{specify individual coordinates for rectangle's upper-left }
{ and bottom-right points, where the points are arrays}
shipRect.topLeft.vh[v] := 20; shipRect.topLeft.vh[h] := 20;
   shipRect.botRight.vh[v] := 70; shipRect.botRight.vh[h] := 70;
```

As described in the chapter “QuickDraw Drawing” in this book, many calculations and graphics operations can be performed on rectangles.

Note

If the bottom coordinate of a rectangle is equal to or less than the top, or the right coordinate is equal to or less than the left, the rectangle is an empty rectangle, one that contains no data. ◆

Regions

The data structure for a region consists of two fixed-length fields followed by a variable-length field:

```
TYPE Region =
   RECORD
      rgnSize:  Integer;{size in bytes}
      rgnBBox:  Rect;   {enclosing rectangle}
      {more data if region is not rectangular}
   END;
```

The `rgnSize` field contains the size, in bytes, of the region. The maximum size is 32 KB when using basic QuickDraw (and 64 KB when using Color QuickDraw). The `rgnBBox` field is a rectangle that completely encloses the region.

The simplest region is a rectangle. In this case, the `rgnBBox` field defines the entire region, and there's no optional region data. For rectangular regions (or empty regions), the `rgnSize` field contains 10. The data for more complex regions is stored in a proprietary format.

As described in the chapter "QuickDraw Drawing" in this book, you can gather an arbitrary set of spatially coherent points into a region and rapidly perform complex manipulations and calculations on them.

The Black-and-White Drawing Environment: Basic Graphics Ports

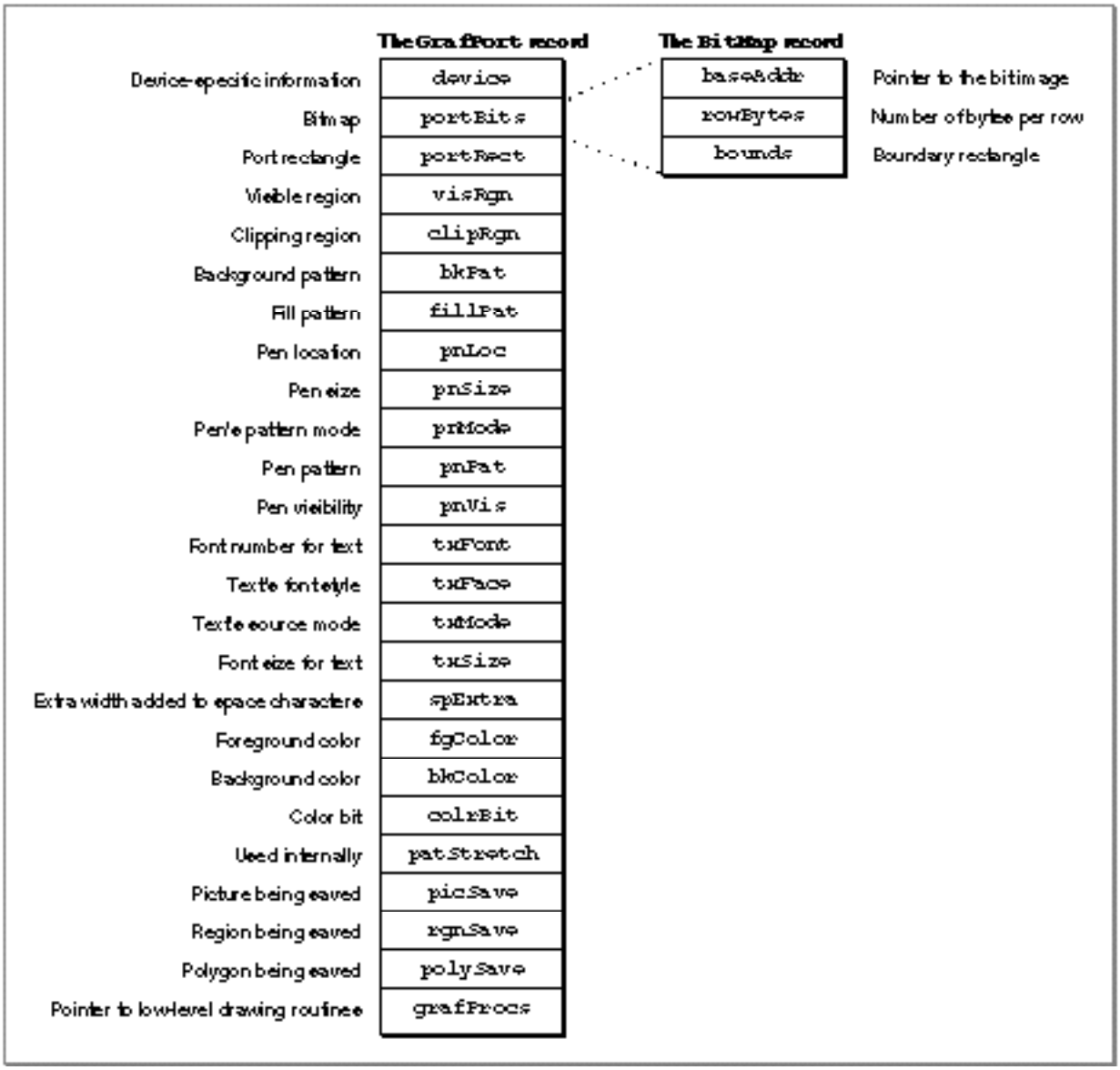
A graphics port is a complete drawing environment that defines where and how graphics operations take place. You can have many graphics ports open at once; each one has its own local coordinate system, drawing pattern, background pattern, pen size and location, font and font style, and bitmap or pixel map (for a color graphics port). You can quickly switch from one graphics port to another.

As described in the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*, the Window Manager incorporates a graphics port in each window record it creates. Similarly, the Printing Manager (described in the chapter "Printing Manager" in this book) incorporates a graphics port in each print record it creates. You can also use the `NewGWorld` function to create graphics ports that are not in a window, and hence not visible on a screen. As described in the chapter "Offscreen Graphics Worlds" in this book, such offscreen graphics worlds are useful for preparing images for display; when the image is ready, you can quickly copy it to an onscreen graphics port.

There are two kinds of graphics ports: the black-and-white, basic graphics port based on the data structure of type `GrafPort`, and the color graphics port based on the data structure of type `CGrafPort` (used only with Color QuickDraw). The basic graphics port is discussed here; the color graphics port is discussed in the chapter "Color QuickDraw." (Using the basic eight-color system described in the chapter "QuickDraw Drawing," you can also use a basic graphics port to display eight predefined colors.)

The `GrafPort` record is diagrammed in Figure 2-1. Some aspects of its contents are discussed after the figure; see page 2-30 for a complete description of the record fields. Your application should not directly set any fields of a `GrafPort` record; instead you should use QuickDraw routines to manipulate them.

Figure 2-1 The `GrafPort` record and the `BitMap` record

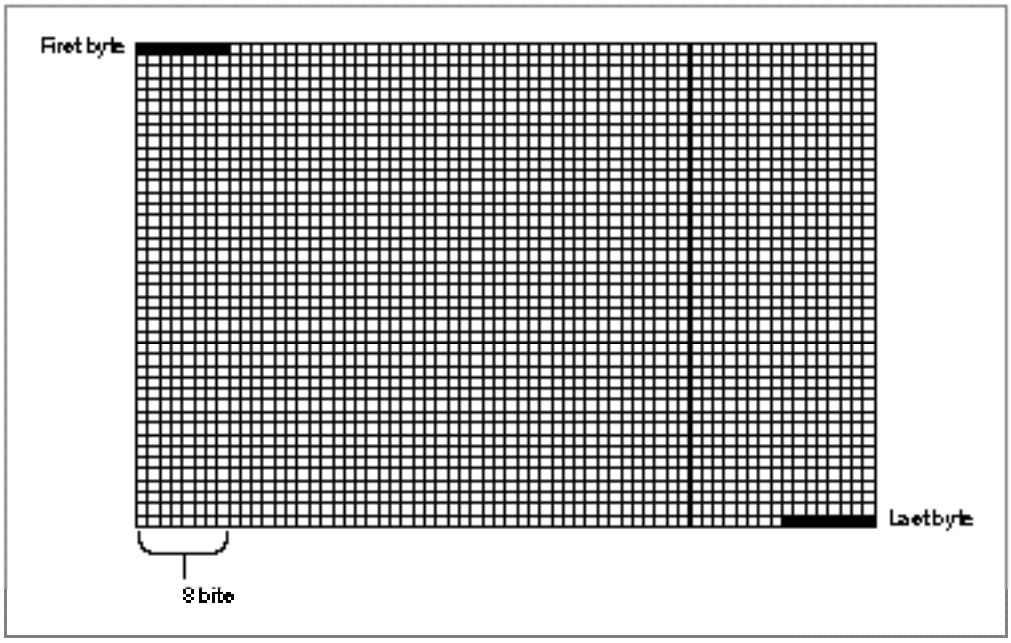


Bitmaps

The `portBits` field of a `GrafPort` record contains the **bitmap,** a data structure of type `BitMap` that defines a black-and-white physical bit image in terms of the QuickDraw coordinate plane. The structure of a bitmap is illustrated in Figure 2-1.

The `baseAddr` field of the `BitMap` record contains a pointer to the beginning of the bit image. (There can be several bitmaps pointing to the same bit image, each imposing its own coordinate system on it.) A **bit image** is a collection of bits in memory that form a grid. To visualize the relationship between the bits in memory and the bits in an image, take a sequence of words in memory and lay them end to end so that bit 15 of the lowest-numbered word is on the left and bit 0 of the highest-numbered word is on the far right. Then take this line of bits and divide it, on word boundaries, into a number of equal-size rows. Stack these rows vertically so that the first row is on the top and the last row is on the bottom. The result is a matrix like the one shown in Figure 2-2—rows and columns of bits, with each row containing the same number of bytes. A bit image can be any length that's a multiple of the row's width in bytes.

Figure 2-2 A bit image



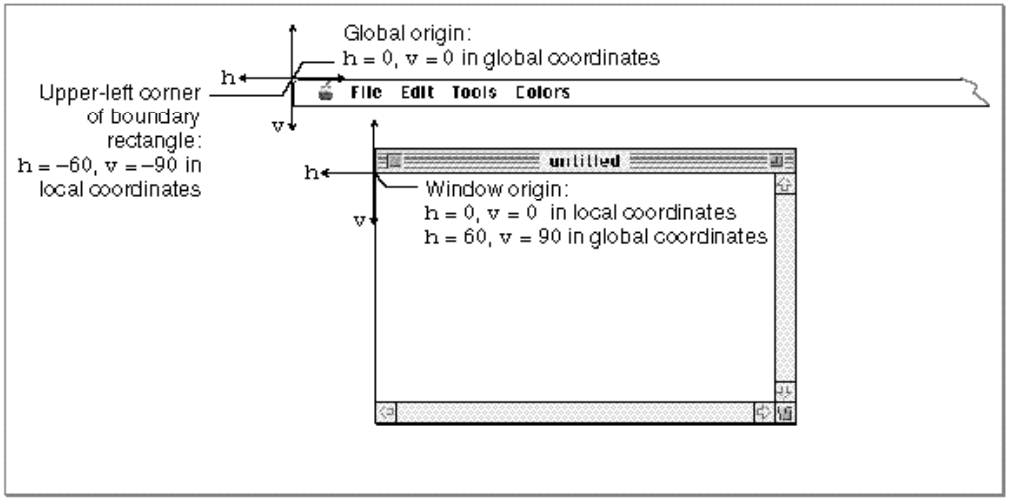
The screen itself is one large visible bit image. On a Macintosh Classic, for example, the screen is a 342-by-512 bit image, with a row width of 64 bytes. These 21,888 bytes of memory are displayed as a matrix of 175,104 pixels on the screen; each bit corresponds to one screen pixel. If a bit's value is 0, its screen pixel is white; if the bit's value is 1, it is black. (Color QuickDraw can work with images that store more than 1 bit for each screen pixel. Such images are called *pixel images*; they are described in the chapter "Color QuickDraw" in this book.)

The `rowBytes` field of the bitmap contains the width of a row of the image in bytes. A bitmap must always begin on a word boundary and contain an integral number of words in each row. The value of the `rowBytes` field must be less than $4000.

The `bounds` field is the bitmap's **boundary rectangle,** which serves two purposes. First, it links the local coordinate system of a graphics port to QuickDraw's global coordinate system. Second, it defines the area of an image into which QuickDraw can draw.

The coordinates of the upper-left corner of the boundary rectangle define the distance from the origin of the graphics port's local coordinate system to the origin of QuickDraw's global coordinate system. In this way, the boundary rectangle links the local coordinate system of a graphics port to QuickDraw's global coordinate system. For example, by subtracting the vertical and horizontal coordinates of the upper-left corner of the boundary rectangle from any other point local to the graphics port, you convert that point into global coordinates. By comparing the origin of a window to the origin of the main screen, Figure 2-3 illustrates the relationship of the boundary rectangle's local coordinate system to QuickDraw's global coordinate system.

Figure 2-3 Relationship of the boundary rectangle and the port rectangle to the global coordinate system



The origin of the local coordinate system is defined by the upper-left corner of the port rectangle for the graphics port. (The port rectangle, as described in "The Graphics Port Drawing Area" on page 2-11, is specified in the `portRect` field of the `GrafPort` record.) In a graphics port for a window, this point is called the **window origin,** and it marks the upper-left corner of a window's content region. As shown in Figure 2-3, this point usually has horizontal and vertical coordinates of 0 in the local coordinate system.

The origin for the global coordinate system has horizontal and vertical coordinates of 0 in the global coordinate system, and, as shown in Figure 2-3, this point lies at the upper-left corner of the main screen.

By default, QuickDraw assigns the entire main screen as the boundary rectangle for a bitmap. Therefore, the local coordinates of the upper-left corner of the boundary rectangle reflect the distance from the window origin to the screen origin. In Figure 2-3, for example, the upper-left corner of the boundary rectangle has a horizontal coordinate of –60 and a vertical coordinate of –90 in the local coordinate system because the window origin has a horizontal coordinate of 60 and a vertical coordinate of 90 in the global coordinate system.

The boundary rectangle defines the area of an image into which QuickDraw can draw. The upper-left corner of the boundary rectangle is aligned around the first bit in the bit image. The width of the boundary rectangle determines how many bits of one row are logically owned by the bitmap. This width must not exceed the number of bits in each row of the bit image (although the width may be smaller than the number of bits in each row).

The height of the boundary rectangle determines how many rows of the bit image are logically owned by the bitmap. The number of rows enclosed by the boundary rectangle must not exceed the number of rows in the bit image (although the number of rows enclosed by the boundary rectangle may be fewer than those in the bit image).

Normally, the boundary rectangle exactly encloses the bit image. If the rectangle is smaller than either dimension of the image, the rightmost bits in each row, or the last rows in the image, or both, are not considered part of the bitmap. All drawing that QuickDraw does in a bitmap is clipped to the edges of the boundary rectangle—bits (and their corresponding pixels) that lie outside the rectangle are unaffected by drawing operations.

The bitmap may be changed to point to a different bit image in memory. All graphics routines work in exactly the same way regardless of whether their effects are visible on the screen. Your application can, for example, prepare an image to be printed on a printer without ever displaying the image on the screen (as described in the chapter “Printing Manager” in this book), or it can prepare an image in an offscreen graphics world before transferring it to the screen (as described in the chapter “Offscreen Graphics Worlds” in this book).

The Graphics Port Drawing Area

Several fields in the `GrafPort` record define your application’s drawing area.

The `portRect` field denotes the port rectangle that defines a subset of the bitmap to be used for drawing. All drawing done by your application occurs inside the port rectangle. As explained in the previous section, the boundary rectangle defines the local coordinate system used by the port rectangle. The port rectangle usually falls within the bitmap’s boundary rectangle, but it’s not required to do so.

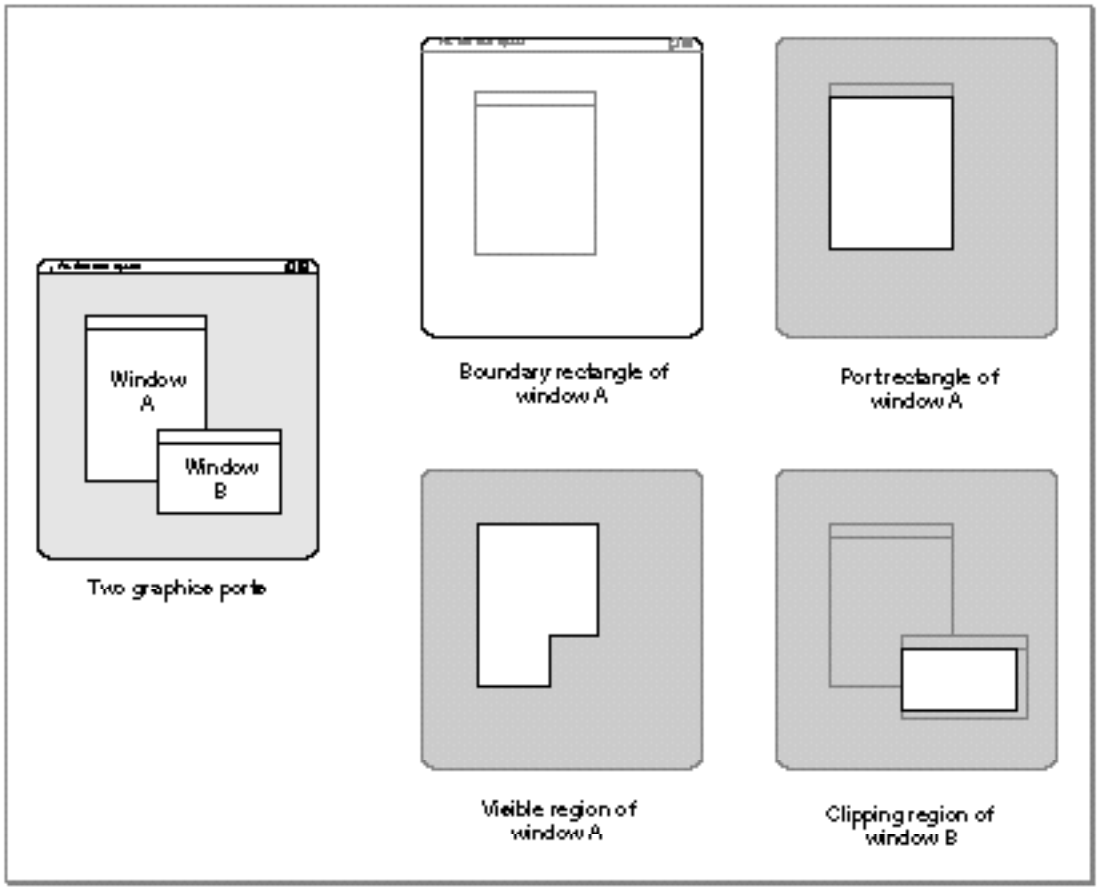
The `visRgn` field designates the visible region of the graphics port. The visible region is the region of the graphics port that’s actually visible on the screen. The visible region is manipulated by the Window Manager. For example, if the user moves one window in front of another, the Window Manager logically removes the area of overlap from the

visible region of the window in back. When you draw into the back window, whatever's being drawn is clipped to the visible region so that it doesn't run over onto the front window.

The `clipRgn` field specifies the graphics port's **clipping region,** which you can use to limit drawing to any region within the port rectangle. The initial clipping region of a graphics port is an arbitrarily large rectangle: one that covers the entire QuickDraw coordinate plane. You can set the clipping region to any arbitrary region, to aid you in drawing inside the graphics port. If, for example, you want to draw a half-circle on the screen, you can set the clipping region to half of the square that would enclose the whole circle, and then draw the whole circle. Only the half within the clipping region is actually drawn in the graphics port.

All drawing in a graphics port occurs in the intersection of the graphics port's boundary rectangle and its port rectangle, and, within that intersection, all drawing is cropped to the graphics port's visible region and its clipping region. No drawing occurs outside the intersection of the port rectangle, the visible region, and the clipping region. Figure 2-4 illustrates several of the previously described fields of the `GrafPort` record.

Figure 2-4 Comparing the boundary rectangle, port rectangle, visible region, and clipping region



As shown in this figure, QuickDraw assigns the entire screen as the boundary rectangle for window A. This boundary rectangle shares the same local coordinate system as the port rectangle for window A. Although not shown in this figure, the upper-left corner—that is, the window origin—of this port rectangle has a horizontal coordinate of 0 and a vertical coordinate of 0, whereas the upper-left corner for window A's boundary rectangle has a horizontal coordinate of –40 and a vertical coordinate of –40.

In this figure, to avoid drawing over scroll bars when drawing into window B, the application that created that window has defined a clipping region that excludes the scroll bars.

Graphics Port Bit Patterns

The `bkPat` and `fillPat` fields of a `GrafPort` record contain patterns used by certain QuickDraw routines. The `bkPat` field contains the background pattern that's used when an area is erased or when bits are scrolled out of it. When asked to fill an area with a specified pattern, QuickDraw stores the given pattern in the `fillPat` field and then calls a low-level drawing routine that gets the pattern from that field.

Bit patterns—which are usually black and white, although any two colors can be used on a color screen—are described in the chapter "QuickDraw Drawing" in this book; patterns with colors at any pixel depth, called *pixel patterns*, are described in the chapter "Color QuickDraw" in this book.

The Graphics Pen

The `pnLoc`, `pnSize`, `pnMode`, `pnPat`, and `pnVis` fields of a graphics port deal with the graphics pen. Each graphics port has one and only one such pen, which is used for drawing lines, shapes, and text. The pen has four characteristics: a location, a size (height and width), a drawing mode, and a drawing pattern. The routines for determining and changing these four characteristics are described in the chapter "QuickDraw Drawing."

Text in a Graphics Port

The `txFont`, `txFace`, `txMode`, `txSize`, and `spExtra` fields of a graphics port determine how text is drawn—the typeface, font style, and font size of characters and how they are placed in the bit image. QuickDraw can draw characters as quickly and easily as it draws lines and shapes, and in many prepared typefaces. The characters may be drawn in any size and font style (that is, with stylistic variations such as bold, italic, and underline). Text is drawn with the base line positioned at the pen location.

For information on using text in your application, including how to use the QuickDraw routines that manipulate text characteristics stored in a graphics port, see *Inside Macintosh: Text*.

The Limited Colors of a Basic Graphics Port

The `fgColor`, `bkColor`, and `colrBit` fields contain values for drawing in the **eight-color system** available with basic QuickDraw. Although limited to eight predefined colors, this system has the advantage of being compatible across all Macintosh platforms. The `fgColor` field contains the graphics port's foreground color, and `bkColor` contains its background color. The `colrBit` field tells the color imaging software which plane of the color picture to draw into.

These colors are recorded when drawing into a QuickDraw picture (described in the chapter "Pictures" in this book)—for example, drawing a line with a red foreground color stores a red line in the picture—but these colors cannot be stored in a bitmap. The basic graphics port's color drawing capabilities are discussed in the chapter "QuickDraw Drawing."

Other Fields

The `patStretch` field is used during printing to expand patterns if necessary. Your application should not change the value of this field.

The `picSave`, `rgnSave`, and `polySave` fields reflect the states of picture, region, and polygon definitions, respectively. To define a region, for example, you open it, call routines that draw it, and then close it. The chapter "QuickDraw Drawing" describes in detail how to use pictures, regions, and polygons to draw into a graphics port.

Finally, the `grafProcs` field may point to a special data structure that your application can store into if you want to customize QuickDraw drawing routines or use QuickDraw in other specialized ways, as described in the chapter "QuickDraw Drawing."

Using Basic QuickDraw

To create a basic QuickDraw drawing environment, you generally

- initialize QuickDraw
- create one or more graphics ports—typically, by using the Window Manager or the `NewGWorld` function
- set a current graphics port whenever your application has multiple graphics ports into which it can draw
- use the coordinate system—local or global—appropriate for the QuickDraw or Macintosh Toolbox routine you wish to use next
- move the document's bit image in relation to the port rectangle of the graphics port when scrolling through a document in a window

These tasks are explained in greater detail in the rest of this chapter. After performing these tasks, your application can draw into the current graphics port, as described in the next chapter, "QuickDraw Drawing."

System 7 added new features to basic QuickDraw that were not available in earlier versions of system software. In particular, System 7 added

- the capability to work with the offscreen graphics worlds described in the chapter "Offscreen Graphics Worlds"
- support for the `OpenCPicture` function to create—and the ability to display—the extended version 2 pictures described in the chapter "Pictures"
- additional capabilities to the `CopyBits` procedure as described in the chapter "QuickDraw Drawing"
- support for the Color QuickDraw routines `RGBForeColor`, `RGBBackColor`, `GetForeColor`, and `GetBackColor` (which are described in the chapter "Color QuickDraw")
- support for the `DeviceLoop` procedure (described in the chapter "Graphics Devices" in this book), which provides your application with information about the current device's pixel depth and other attributes
- support for the Picture Utilities, as described in the chapter "Pictures" in this book (however, when collecting color information on a computer running only basic QuickDraw, the Picture Utilities return `NIL` instead of handles to `Palette` and `ColorTable` records)

Before using these capabilities, you should make sure they are available by using the `Gestalt` function with the `gestaltSystemVersion` selector. Test the low-order word in the `response` parameter; if the value is $0700 or greater, then the System 7 features of basic QuickDraw are supported.

You can test whether a computer supports only basic QuickDraw with no Color QuickDraw support by using the `Gestalt` function with the selector `gestaltQuickDrawVersion`. The `Gestalt` function returns a 4-byte value in its `response` parameter; the low-order word contains QuickDraw version data. If `Gestalt` returns the value represented by the constant `gestaltOriginalQD`, then Color QuickDraw is not supported.

The `Gestalt` function is described in the chapter "Gestalt Manager" of *Inside Macintosh: Operating System Utilities*.

Initializing Basic QuickDraw

Call the `InitGraf` procedure to initialize QuickDraw at the beginning of your program, before initializing any other parts of the Toolbox, as shown in the application-defined procedure `DoInit` in Listing 2-1. The `InitGraf` procedure initializes both basic QuickDraw and, on computers that suppport it, Color QuickDraw.

Listing 2-1 Initializing QuickDraw

```
PROCEDURE DoInit;
BEGIN
 DoSetUpHeap;        {perform Memory Manager initialization here}
 InitGraf(@thePort); {initialize QuickDraw}
 InitFonts;          {initialize Font Manager}
 InitWindows;        {initialize Window Manager & other Toolbox }
                     { managers here}
                     {perform all other initializations here}
 InitCursor;         {set cursor to an arrow instead of a clock}
END; {of DoInit}
```

When your application starts up, the Finder sets the cursor to a wristwatch; this indicates that a lengthy operation is in progress. See the chapter "Cursor Utilities" in this book for information about changing the cursor when appropriate.

Creating Basic Graphics Ports

All graphics operations are performed in graphics ports. Before a basic graphics port can be used, it must be allocated and initialized with the `OpenPort` procedure. Normally, you don't call `OpenPort` yourself. In most cases your application draws into a window you've created with the `GetNewWindow` or `NewWindow` function (or, for color windows, `GetNewCWindow` or `NewCWindow`), or it draws into an offscreen graphics world created with the `NewGWorld` function. These Window Manager functions (described in the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*) and the `NewGWorld` function (described in the chapter "Offscreen Graphics Worlds" in this book) call `OpenPort` to create a basic graphics port. See the description of the `OpenPort` procedure on page 2-38 for a table of initial values for a basic graphics port.

Listing 2-2 shows a simplified application-defined procedure called `DoNew` that uses the Window Manager function `GetNewWindow` to create a basic graphics port for computers that do not support color. The `GetNewWindow` function returns a window pointer, which is defined to be a pointer to graphics port.

Listing 2-2 Using the Window Manager to create a basic graphics port

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

You can allow `GetNewWindow` to allocate the memory for your window record and its associated basic graphics port. You can maintain more control over memory use, however, by allocating the memory yourself from a block allocated for such purposes during your own initialization routine, and then passing the pointer to `GetNewWindow`, as shown in Listing 2-2.

When you call the `CloseWindow` or `DisposeWindow` procedure to close or dispose of a window, the Window Manager disposes of the graphics port's regions by calling the `ClosePort` procedure. If you use the `CloseWindow` procedure, you also dispose of the window record containing the graphics port by calling the Memory Manager procedure `DisposePtr`.

For detailed information about managing windows, see the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*. For detailed information about managing memory, see *Inside Macintosh: Memory*.

Setting the Graphics Port

Before drawing into the window, Listing 2-2 calls the `SetPort` procedure to make the window the current graphics port. If your application draws into more than one graphics port, you can call `SetPort` to set the graphics port into which you want to draw. At times you may need to preserve the current graphics port. As shown in Listing 2-3, you can do this by calling the `GetPort` procedure to save the current graphics port, `SetPort` to set the graphics port you want to draw in, and then `SetPort` again when you need to restore the previous graphics port. (The procedures also work with color graphics ports.)

Listing 2-3 Saving and restoring a graphics port

```
PROCEDURE DrawInPort (thePort: GrafPtr);
VAR
   origPort: GrafPtr;
BEGIN
   GetPort(origPort);      {save the original port}
   SetPort(thePort);       {set a new port}
   DoDrawWindow(thePort);  {draw into the new port}
   SetPort(origPort);      {restore the original port}
END;
```

In this example, the application calling `DrawInPort` may need to temporarily turn an inactive window into the current graphics port for updating purposes. After drawing into the inactive window, `DrawInPort` makes the user's active window the current graphics port again.

Note

When your application runs in Color QuickDraw or uses offscreen graphics worlds, it should use the `GetGWorld` procedure instead of `GetPort`, and it should use the `SetGWorld` procedure instead of `SetPort`. These procedures save and restore the current graphics port for basic and color graphics ports as well as offscreen graphics worlds. See the chapter "Offscreen Graphics Worlds" in this book for more information. ◆

Switching Between Global and Local Coordinate Systems

Each graphics port has its own local coordinate system. Some Toolbox routines return or expect points that are expressed in the global coordinate system, while others use local coordinates. Sometimes you need to use the `GlobalToLocal` procedure to convert global coordinates to local coordinates, and sometimes you need the `LocalToGlobal` procedure for the reverse operation. For example, when the Event Manager function `WaitNextEvent` reports an event, it gives the cursor location (also called the *mouse location*) in global coordinates; but when you call the Control Manager function `FindControl` to find out whether the user clicked a control in one of your windows, you pass the cursor location in local coordinates, as shown in Listing 2-4. (The Event Manager and the Control Manager are described in *Inside Macintosh: Macintosh Toolbox Essentials*.)

Listing 2-4 Changing global coordinates to local coordinates

```
PROCEDURE DoControlClick (window: WindowPtr; event: EventRecord);
VAR
mouse:      Point;
control:    ControlHandle;
part:       Integer;
windowType: Integer;
BEGIN
      SetPort(window);
      mouse := event.where;   {save the cursor location}
      GlobalToLocal(mouse);   {convert to local coordinates}
      part := FindControl(mouse, window, control);
      CASE part OF
         inButton:   {mouse-down in OK button}
            DoOKButton(mouse, control);
         inCheckBox: {mouse-down in checkbox}
            DoCheckBox(mouse, control);
         OTHERWISE
         ;
      END;  {of CASE for control part codes}
END; {of DoControlClick}
```

Scrolling the Pixels in the Port Rectangle

If your application scrolls a document in a window, your application can use the `ScrollRect` procedure to shift the pixels currently displayed for that document, and then it can use the `SetOrigin` procedure to adjust the window's local coordinate system for drawing a new portion of the document inside the update region of the window.

Scrolling a document in response to the user's manipulation of a scroll bar requires you to use the Control Manager, the Window Manager, and the File Manager in addition to QuickDraw. The chapter "Control Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* provides a thorough explanation of how to scroll through documents. An overview of the necessary tasks is provided here.

A window record contains a graphics port in its first field, and the Window Manager uses the port rectangle of the graphics port as the content area of the window. This allows you to use the QuickDraw routines `ScrollRect` and `SetOrigin`—which normally operate on the port rectangle of a graphics port—to manipulate the content area of the window.

The left side of Figure 2-5 illustrates a case where the user has just opened an existing document, and the application displays the top of the document. In this example, the document consists of 35 lines of monospaced text, and the line height throughout is 10 pixels. Therefore, the document is 350 pixels long. The application stores the document in a document record of its own creation. This document record assigns its own coordinate system to the document. When the user first opens the document, the upper-left point of the graphics port's port rectangle (the window origin) is identical to the upper-left point of the document record's own coordinate system: both have a horizontal coordinate of 0 and a vertical coordinate of 0.

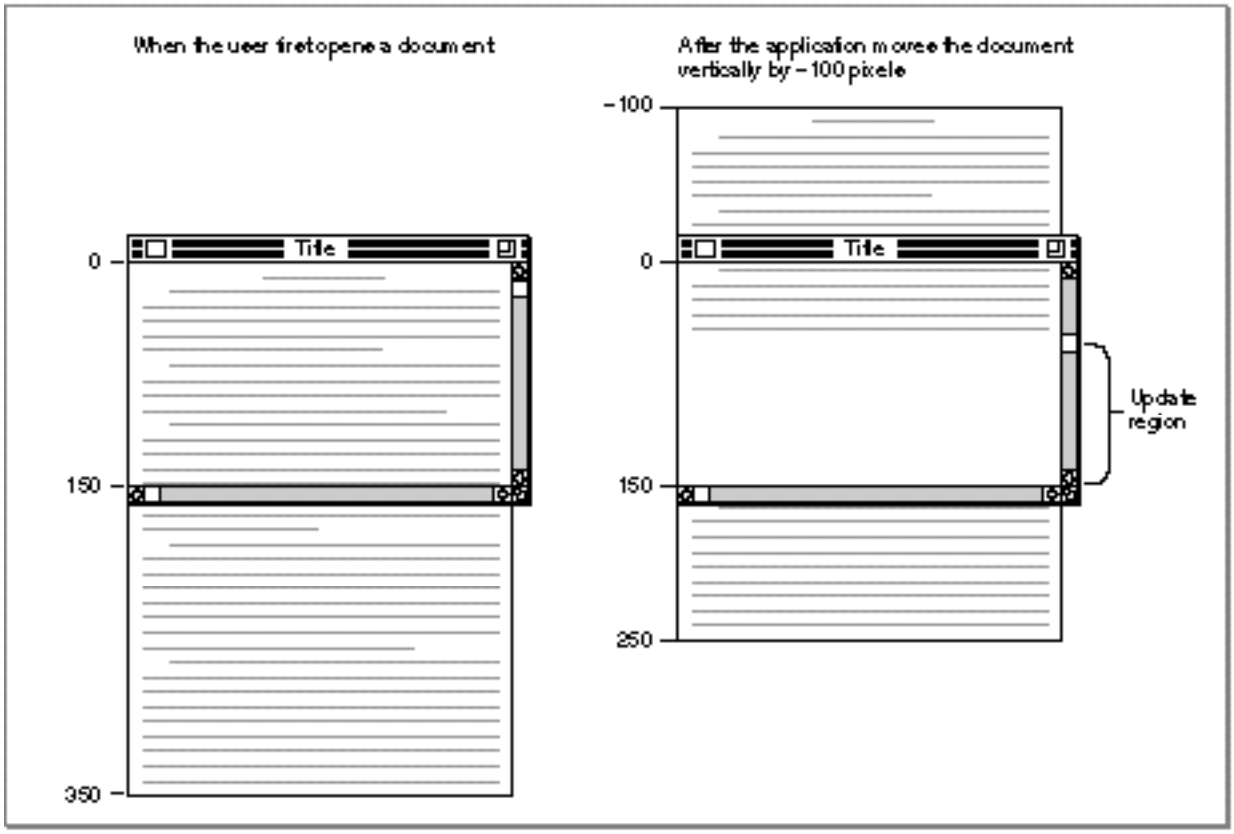
In this example, the content area—that is, the port rectangle—of the window displays 15 lines of text, which amount to 150 pixels.

Imagine that the user drags the scroll box part way down the vertical scroll bar. Because the user wishes to scroll down, the application must move the document up so that more of the bottom of the document shows. Moving a document *up* in response to a user request to scroll *down* requires a scrolling distance with a *negative* value. (Likewise, moving a document *down* in response to a user request to scroll *up* requires a scrolling distance with a *positive* value.)

Using the Control Manager functions `FindControl`, `TrackControl`, and `GetControlValue`, the application in this example determines that it must move the document up by 100 pixels—that is, by a scrolling distance of –100 pixels.

The application uses the QuickDraw procedure `ScrollRect` to shift the pixels currently displayed in the port rectangle of the window by a distance of –100 pixels. This moves the portion of the document displayed in the window upward by 100 pixels (that is, by 10 lines); 5 lines that were previously displayed at the bottom of the window now appear at the top of the window, and the application adds the rest of the window to an update region for later updating.

Figure 2-5 Moving a document relative to its window



The `ScrollRect` procedure doesn't change the coordinate system of the graphics port for the window; instead it moves the pixels in a specified rectangle (in this case, the port rectangle) to new coordinates that are still in the graphics port's local coordinate system. For purposes of updating the window, you can think of this as changing the coordinates used by the application's document record, as illustrated in the right side of Figure 2-5.

The `ScrollRect` procedure takes four parameters: a rectangle to scroll, a horizontal distance to scroll, a vertical distance to scroll, and a region handle. Typically, when specifying the rectangle to scroll, your application passes a value representing the port rectangle (that is, the window's content region) minus the scroll bar regions, as shown in Listing 2-5.

Listing 2-5 Using `ScrollRect` to scroll the bits displayed in the window

```
PROCEDURE DoGraphicsScroll (window: WindowPtr;
                            hDistance,vDistance: Integer);
VAR
   myScrollRect: Rect;
   updateRegion: RgnHandle;
BEGIN
   {initially, use the window's portRect as the rectangle to scroll:}
   myScrollRect := window^.portRect;
   {subtract vertical and horizontal scroll bars from rectangle}
   myScrollRect.right := myScrollRect.right - 15;
   myScrollRect.bottom := myScrollRect.bottom - 15;
   updateRegion := NewRgn;    {always initialize the update region}
   ScrollRect(myScrollRect, hDistance, vDistance, updateRegion);
   InvalRgn(updateRegion);
   DisposeRgn(updateRegion);
END; {of DoGraphicsScroll}
```

The pixels that `ScrollRect` shifts outside of the rectangle specified by the `myScrollRect` variable are not drawn on the screen, and the bits they represent are not saved—it is your application's responsibility to keep track of this data.

The `ScrollRect` procedure shifts the image displayed inside the port rectangle by a distance of `hDistance` pixels horizontally and `vDistance` pixels vertically; when the `DoGraphicsScroll` procedure passes positive values in these parameters, `ScrollRect` shifts the pixels in `myScrollRect` to the right and down, respectively. This is appropriate when the user intends to scroll left or up because, when the application finishes updating the window, the user sees more of the left and top of the document, respectively. (Remember: to scroll up or left, move the pixels down or right, both of which are in the positive direction.)

When `DoGraphicsScroll` passes negative values in these parameters, `ScrollRect` shifts the pixels in `myScrollRect` to the left or up. This is appropriate when the user intends to scroll right or down because, when the application finishes updating the window, the user sees more of the right and the bottom of the document. (Remember: to scroll down or right, move the bit image up or left, both of which are in the negative direction.)

In Figure 2-5, the application determines a vertical scrolling distance of –100, which it passes in the `vDistance` parameter as shown here:

```
ScrollRect(myScrollRect, 0, –100, updateRegion);
```

If, however, the user were to move the scroll box back to the beginning of the document at this point, the application would determine that it has a distance of 100 pixels to scroll up, and it would therefore pass a positive value of 100 in the `vDistance` parameter.

By creating an update region for the window, `ScrollRect` forces an update event. After using `ScrollRect` to move the bit image that already exists in the window, the application must use its own window-updating code to draw pixels in the update region of the window. (See the chapter "QuickDraw Drawing" in this book for information about drawing into a window.)

As previously explained, `ScrollRect` in effect changes the coordinates of the application's document record relative to the local coordinates of the port rectangle. In terms of the graphics port's local coordinate system, the upper-left corner of the document now has a vertical coordinate of –100, as shown on the right side of Figure 2-5 on page 2-21. To facilitate updating the window, the application uses the `SetOrigin` procedure to change the window origin of the port rectangle so that the application can treat the upper-left corner of the document as again having a local horizontal coordinate of 0 and a local vertical coordinate of 0.

The `SetOrigin` procedure takes two parameters: the first is a new horizontal coordinate for the upper-left corner of the port rectangle, and the second is a new vertical coordinate for the upper-left corner of the port rectangle.

Any time you are ready to update a window (for example, after scrolling it), you can use the Control Manager function `GetControlValue` to determine the current setting of the horizontal scroll bar, and you can pass this value to `SetOrigin` as the new horizontal coordinate for the window origin. Then use `GetControlValue` to determine the current setting of the vertical scroll bar. Pass this value to `SetOrigin` as the new vertical coordinate for the window origin. Using `SetOrigin` in this fashion lets you treat the upper-left corner of the document as always having a horizontal coordinate of 0 and a vertical coordinate of 0 when you update (that is, redraw) the document within a window.

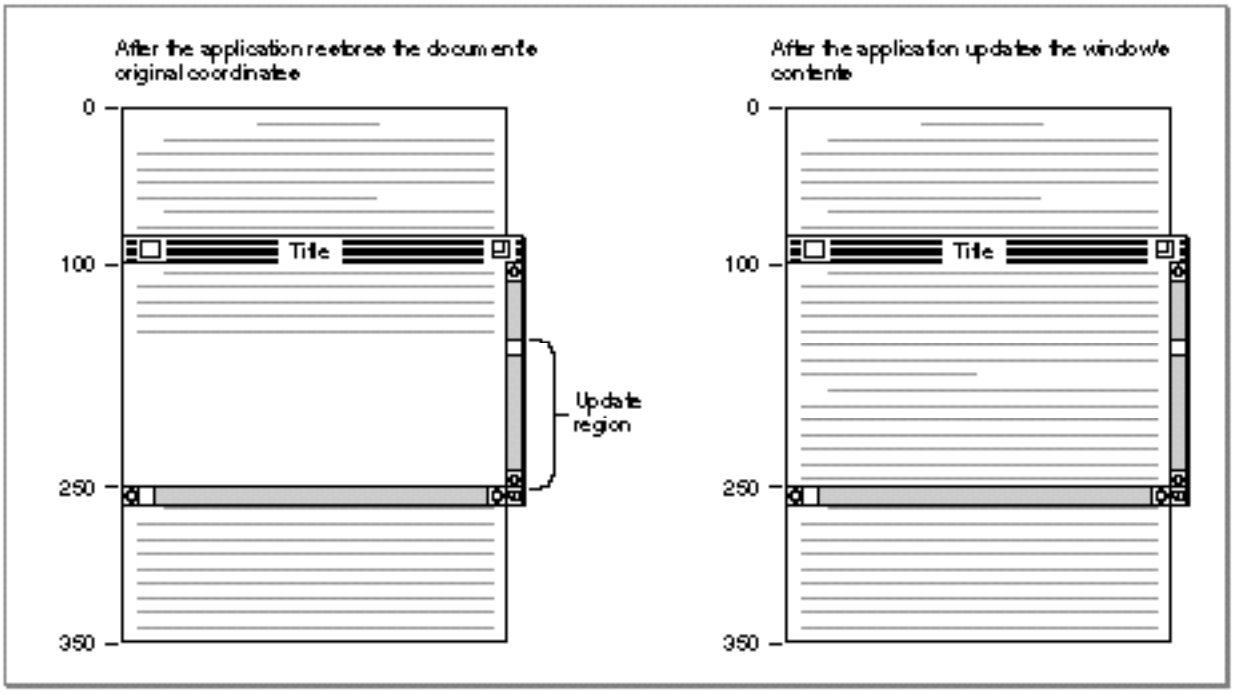
For example, after the user manipulates the vertical scroll bar to move (either up or down) to a location 100 pixels from the top of the document, the application makes the following call:

```
SetOrigin(0, 100);
```

Although the scrolling distance in Figure 2-5 is –100, which is relative, the current setting for the scroll bar on the right side of the figure is now at 100.

The left side of Figure 2-6 shows how the application uses the `SetOrigin` procedure to move the window origin so that the upper-left corner of the document now has a horizontal coordinate of 0 and a vertical coordinate of 0 in the graphics port's local coordinate system. This restores the coordinates that the application originally assigned to the document in its document record and makes it easier for the application to draw in the update region of the window.

Figure 2-6 Updating the contents of a scrolled window

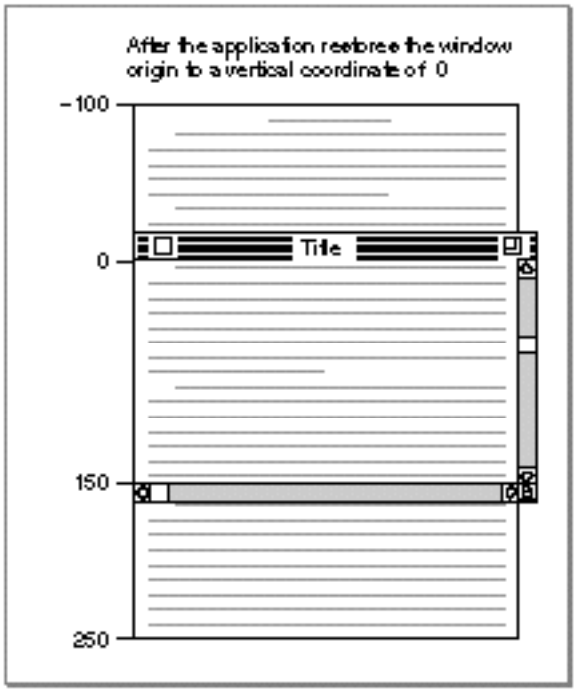


After restoring the document's original coordinates, the application updates the window, as shown on the right side of Figure 2-6. The application draws lines 16 through 24, which it stores in its own document record as beginning at a vertical coordinate of 160 and ending at a vertical coordinate of 250.

To review what has happened up to this point: the user has dragged the scroll box down the vertical scroll bar; the application determines that this amounts to a scroll distance of –100 pixels; the application passes this distance to `ScrollRect`, which shifts the document displayed in the window upward by 100 pixels and creates an update region for the rest of the window; the application passes the vertical scroll bar's current setting (100 pixels) in a parameter to `SetOrigin` so that the upper-left corner of the document has a horizontal coordinate of 0 and a vertical coordinate of 0 in the local coordinate system of the graphics port; and, finally, the application draws the text in the update region of the window.

However, the window origin of the port rectangle cannot be left at the point with a horizontal value of 0 and a vertical value of 100; instead, the application must use `SetOrigin` to reset it to a horizontal coordinate of 0 and a vertical coordinate of 0 after performing its own drawing, because the Window Manager and Control Manager always assume the window's upper-left point has a horizontal coordinate of 0 and a vertical coordinate of 0 when they draw in a window. Figure 2-7 shows how the application uses `SetOrigin` to set the upper-left corner of the port rectangle back to a horizontal coordinate of 0 and a vertical coordinate of 0 at the conclusion of its window-updating routine.

Figure 2-7 Restoring the window origin of the port rectangle to a horizontal coordinate of 0 and a vertical coordinate of 0



This example illustrates how to use `SetOrigin` to offset the port rectangle's coordinate system so that you can treat objects in a document as fixed in the document's own coordinate space. Alternatively, it's possible to leave the coordinate system for the graphics port fixed and instead offset the items in a document by the amount equal to the scroll bar settings. The `OffsetRect` and `OffsetRgn` procedures (which are described in the chapter "QuickDraw Drawing"), the `SubPt` procedure (described on page 2-53), and the `AddPt` procedure (described on page 2-52) are useful if you pursue this approach. However, it is recommended that you use `SetOrigin` instead.

IMPORTANT
For optimal performance and future compatibility, you should use the `SetOrigin` procedure when reconciling document coordinate space with the local coordinate system of your graphics port. ▲

The `SetOrigin` procedure does not move the window's clipping region. If you use clipping regions in your windows, use the `GetClip` procedure (described on page 2-47) to store your clipping region immediately after your first call to `SetOrigin`. Before calling your own window-drawing routine, use the `ClipRect` procedure (described on page 2-49) to define a new clipping region—to avoid drawing over your scroll bars, for example. (Listing 3-9 on page 3-29 in the chapter "QuickDraw Drawing" illustrates how to do this.) After calling your own window-drawing routine, use the `SetClip` procedure (described on page 2-48) to restore the original clipping region. You can then call `SetOrigin` again to restore the window origin to a horizontal coordinate of 0 and a vertical coordinate of 0 with your original clipping region intact.

Basic QuickDraw Reference

This section describes the data structures and routines that are specific to basic QuickDraw.

"Data Structures" shows the data structures for a point, rectangle, region, bitmap, and basic graphics port. "Routines" describes basic QuickDraw routines for initializing QuickDraw; opening and closing basic graphics ports; saving and restoring graphics ports; managing bitmaps, port rectangles, and clipping regions; and manipulating points in graphics ports.

Data Structures

This section describes the data structures that represent a point, rectangle, region, bitmap, and basic graphics port.

You use the point (a data structure of type `Point`) to specify a location on the QuickDraw coordinate plane; two points are sufficient to define a rectangle. The rectangle (a data structure of type `Rect`) in turn assigns coordinates to boundaries and images; rectangles also bound graphic objects such as regions and ovals.

The region (a data structure of type `Region`) defines an arbitrary area, such as the visible and clipping regions of a window's graphics port.

The bitmap (a data structure of type `BitMap`) defines a physical bit image in terms of the QuickDraw coordinate plane.

The basic graphics port is a data structure (of type `GrafPort`) upon which your application builds windows.

Point

You use a point, which is a data structure of type `Point`, to specify a location on the QuickDraw coordinate plane. For example, the window origin is specified by the point in the upper-left corner of the port rectangle of a graphics port.

```
TYPE VHSelect = (v,h);
Point =
RECORD
   CASE Integer OF
      0: (v:   Integer:    {vertical coordinate}
          h:   Integer);   {horizontal coordinate}
      1: (vh:  ARRAY[VHSelect] OF Integer);
END;
```

Field descriptions

`v`	The vertical coordinate of the point.
`h`	The horizontal coordinate of the point.
`vh`	A variant definition in which `v` and `h` are array elements.

Note that while the vertical coordinate (`v`) appears first in this data structure, followed by the horizontal coordinate (`h`), the parameters to all QuickDraw routines expect the horizontal coordinate first and the vertical coordinate second.

QuickDraw routines for calculating and changing points are described in “Manipulating Points in Graphics Ports” beginning on page 2-51.

Rect

You can use a rectangle, which is a data structure of type `Rect`, to define areas on the screen and to specify the locations and sizes for various graphics operations. For example, a port rectangle represents the area of a graphics port (described on page 2-30) available for drawing.

The `Rect` data type can be defined by two points or four integers. The two points define the upper-left and lower-right corners of a rectangle; the four integers define the vertical and horizontal coordinates of the two points.

```
TYPE Rect =
RECORD
   CASE Integer OF            {cases: 4 boundaries or 2 corners}
      0: (top:       Integer;    {upper boundary of rectangle}
          left:      Integer;    {left boundary of rectangle}
          bottom:    Integer;    {lower boundary of rectangle}
          right:     Integer);   {right boundary of rectangle}
      1: (topLeft:   Point;      {upper-left corner of rectangle}
          botRight:  Point);     {lower-right corner of rectangle}
END;
```

Field descriptions

`top`	The vertical coordinate of the upper-left point of the rectangle.
`left`	The horizontal coordinate of the upper-left point of the rectangle.
`bottom`	The vertical coordinate of the lower-right point of the rectangle.
`right`	The horizontal coordinate of the lower-right point of the rectangle.
`topLeft`	The upper-left corner of the rectangle.
`botRight`	The lower-right corner of the rectangle.

Note that while the vertical coordinate appears first in this data structure, followed by the horizontal coordinate, the parameters to all QuickDraw routines expect the horizontal coordinate first and the vertical coordinate second.

See the chapter "QuickDraw Drawing" for descriptions of the QuickDraw routines you can use for calculating and manipulating rectangles for drawing purposes.

Region

You can use a region, which is a data structure of type `Region`, to define an arbitrary area or set of areas on the QuickDraw coordinate plane. For example, when scrolling through a window, your application must initialize an update region and pass its handle to the `ScrollRect` procedure (which is described on page 2-43).

The data structure for a region consists of two fixed-length fields followed by a variable-length field.

```
TYPE RgnHandle =  ^RgnPtr;
RgnPtr =          ^Region;
Region =
RECORD
   rgnSize:  Integer;   {size in bytes}
   rgnBBox:  Rect;      {enclosing rectangle}
   {more data if region is not rectangular}
END;
```

Field descriptions

`rgnSize` The region's size in bytes.

`rgnBBox` The rectangle that bounds the region.

The maximum size of a region is 32 KB when using basic QuickDraw, 64 KB when using Color QuickDraw. The simplest region is a rectangle. In this case, the `rgnBBox` field defines the entire region, and there's no optional region data. For rectangular regions (or empty regions), the `rgnSize` field contains 10.

Region data is stored in a proprietary format.

See the chapter "QuickDraw Drawing" for descriptions of the QuickDraw routines you can use for calculating and manipulating regions for drawing purposes.

BitMap

A bitmap, which is a data structure of type `BitMap`, defines a bit image in terms of the QuickDraw coordinate plane. (A bit image is a collection of bits in memory that form a grid; Figure 2-2 on page 2-9 illustrates a bit image.)

A bitmap has three parts: a pointer to a bit image, the row width of that image, and a boundary rectangle that links the local coordinate system of a graphics port to QuickDraw's global coordinate system and defines the area of the bit image into which QuickDraw can draw.

```
TYPE BitMap =
RECORD
   baseAddr:   Ptr;     {pointer to bit image}
   rowBytes:   Integer; {row width}
   bounds:     Rect;    {boundary rectangle}
END;
```

Field descriptions

`baseAddr`	A pointer to the beginning of the bit image.
`rowBytes`	The offset in bytes from one row of the image to the next. The value of the `rowBytes` field must be less than $4000.
`bounds`	The bitmap's boundary rectangle; by default, the entire main screen.

The width of the boundary rectangle determines how many bits of one row are logically owned by the bitmap. (Figure 2-3 on page 2-10 illustrates a boundary rectangle.) This width must not exceed the number of bits in each row of the bit image. The height of the boundary rectangle determines how many rows of the image are logically owned by the bitmap. The number of rows enclosed by the boundary rectangle must not exceed the number of rows in the bit image.

The boundary rectangle defines the local coordinate system used by the port rectangle for a graphics port (described next). The upper-left corner (which for a window is called the *window origin*) of the port rectangle usually has a vertical coordinate of 0 and a horizontal coordinate of 0, although you can use the `SetOrigin` procedure (described on page 2-45) to change the coordinates of the window origin.

GrafPort

A basic graphics port, which is a data structure of type `GrafPort`, defines a complete drawing environment that determines where and how black-and-white graphics operations take place. (Using the basic eight-color system described in the chapter "QuickDraw Drawing," you can also use a basic graphics port to display eight predefined colors.)

All graphics operations are performed in graphics ports. Before a basic graphics port can be used, it must be allocated and initialized with the `OpenPort` procedure. Normally, you don't call `OpenPort` yourself. In most cases your application draws into a window you've created with the `GetNewWindow` or `NewWindow` function (or, for color windows, `GetNewCWindow` or `NewCWindow`), or it draws into an offscreen graphics world created with the `NewGWorld` function. These Window Manager functions (described in the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*) and the `NewGWorld` function (described in the chapter "Offscreen Graphics Worlds" in this book) call `OpenPort` to create a basic graphics port. See the description of the `OpenPort` procedure on page 2-38 for a table of initial graphics port values.

You can have many graphics ports open at once; each one has its own local coordinate system, pen pattern, background pattern, pen size and location, font and font style, and bitmap in which drawing takes place. Using the `SetPort` procedure (described on page 2-42), you can instantly switch from one port to another.

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

```
TYPE GrafPtr = ^GrafPort;
GrafPort =
RECORD
   device:     Integer;    {device-specific information}
   portBits:   BitMap;     {bitmap}
   portRect:   Rect;       {port rectangle}
   visRgn:     RgnHandle;  {visible region}
   clipRgn:    RgnHandle;  {clipping region}
   bkPat:      Pattern;    {background pattern}
   fillPat:    Pattern;    {fill pattern}
   pnLoc:      Point;      {pen location}
   pnSize:     Point;      {pen size}
   pnMode:     Integer;    {pattern mode}
   pnPat:      Pattern;    {pen pattern}
   pnVis:      Integer;    {pen visibility}
   txFont:     Integer;    {font number for text}
   txFace:     Style;      {text's font style}
   txMode:     Integer;    {source mode for text}
   txSize:     Integer;    {font size for text}
   spExtra:    Fixed;      {extra space}
   fgColor:    LongInt;    {foreground color}
   bkColor:    LongInt;    {background color}
   colrBit:    Integer;    {color bit}
   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:  QDProcsPtr; {low-level drawing routines}
END;

WindowPtr = GrafPtr;
```

▲ **WARNING**

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

Field descriptions

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

`portBits` The bitmap (described on page 2-29) that describes the boundary rectangle for the graphics port and contains a pointer to the bit image used by the graphics port.

`portRect` The port rectangle that defines a subset of the bitmap to be used for drawing. All drawing done by the application occurs inside the port rectangle. (In a window's graphics port, the port rectangle is also called the *content region*.) The port rectangle uses the local coordinate system defined by the boundary rectangle in the `portBits` field of the `BitMap` record. The upper-left corner (which for a window is called the *window origin*) of the port rectangle usually has a vertical coordinate of 0 and a horizontal coordinate of 0, although you can use the `SetOrigin` procedure (described on page 2-45) to change the coordinates of the window origin. The port rectangle usually falls within the boundary rectangle, but it's not required to do so.

`visRgn` The region of the graphics port that's actually visible on the screen—that is, the part of the window that's not covered by other windows. By default, the visible region is equivalent to the port rectangle. The visible region has no effect on offscreen images.

`clipRgn` The graphics port's clipping region, an arbitrary region that you can use to limit drawing to any region within the port rectangle. The default clipping region is set arbitrarily large; using the `ClipRect` procedure (described on page 2-49), you have full control over its setting. Unlike the visible region, the clipping region affects the image even if it isn't displayed on the screen.

`bkPat` The background bit pattern that's used by procedures such as `ScrollRect` (described on page 2-43) and `EraseRect` (described in the chapter "QuickDraw Drawing") for filling scrolled or erased areas. Your application can use the `BackPat` procedure (described in the chapter "QuickDraw Drawing") to change the background bit pattern. This pattern, like all other patterns drawn in the graphics port, is always aligned with the port's coordinate system. The upper-left corner of the pattern is aligned with the upper-left corner of the port rectangle, so that adjacent areas of the same pattern blend into a continuous, coordinated pattern. Bit patterns are described in the chapter "QuickDraw Drawing."

`fillPat` The bit pattern that's used when you use a procedure such as `FillRect` to fill an area. Bit patterns are described in the chapter "QuickDraw Drawing."

`pnLoc` The point where QuickDraw will begin drawing the next line, shape, or character. It can be anywhere on the coordinate plane; there are no restrictions on the movement or placement of the pen. The location of the graphics pen is a point in the graphics port's coordinate system, not a pixel in a bit image. The upper-left corner of the pen is at the pen location; the graphics pen hangs below and to the right of this point. You can use the `Move`, `MoveTo`, `Line`, and `LineTo` procedures (described in the chapter "QuickDraw Drawing") to move the location of the graphics pen.

`pnSize` The vertical and horizontal dimensions of the graphics pen. By default, the pen is 1 pixel high by 1 pixel wide; the height and width can range from 0 by 0 to 32,767 by 32,767. If either the pen width or the pen height is 0, the pen does not draw. Heights or widths of less than 0 are undefined. You can use the `PenSize` procedure (described in the chapter "QuickDraw Drawing") to change the value in this field.

`pnMode` The pattern mode—that is, a Boolean operation that determines the how QuickDraw transfers the pen pattern to the bitmap during drawing operations. When the graphics pen draws into a bitmap, QuickDraw first determines what bits in the bit image are affected and then finds their corresponding bits in the pen pattern. QuickDraw then does a bit-by-bit comparison based on the pattern mode, which specifies one of eight Boolean transfer operations to perform. QuickDraw stores the resulting bit in its proper place in the bit image. Pattern modes for a basic graphics port are described in the chapter "QuickDraw Drawing."

`pnPat` A bit pattern that's used like the ink in the pen. As described in the chapter "QuickDraw Drawing," basic QuickDraw uses this pattern when you use the `Line` and `LineTo` procedures to draw lines with the pen, framing procedures such as `FrameRect` to draw shape outlines with the pen, or painting procedures such as `PaintRect` to paint shapes with the pen.

`pnVis` The graphics pen's visibility—that is, whether it draws on the screen. The graphics pen is described in detail in the chapter "QuickDraw Drawing."

`txFont` A font number that identifies the font to be used in the graphics port. The font number 0 represents the system font. (A font is defined as a collection of images that represent the individual characters of the font. A font can consist of up to 255 distinct characters, yet not all characters need to be defined in a single font. In addition, each font contains a missing symbol to be drawn in case of a request to draw a character that's missing from the font.) Fonts are described in detail in *Inside Macintosh: Text*.

`txFace` The font style of the text, with values from the set defined by the `Style` data type, which includes such styles as bold, italic, and shaded. You can apply stylistic variations either alone or in combination. Font styles are described in detail in *Inside Macintosh: Text*.

`txMode` One of three Boolean source modes that determines the way characters are placed in the bit image. This mode functions much like a pattern mode specified in the `pnMode` field: when drawing a character, QuickDraw determines which bits in the bit image are affected, does a bit-by-bit comparison based on the mode, and stores the resulting bits into the bit image. Only three source modes—`srcOr`, `srcXor`, and `srcBic`—should be used for drawing text. See the chapter "QuickDraw Text" in *Inside Macintosh: Text* for more information about QuickDraw's text-handling capabilities.

`txSize` The text size in pixels. The Font Manager uses this information to provide the bitmaps for text drawing. (The Font Manager is described in detail in the chapter "Font Manager" in *Inside Macintosh: Text*.) The value in this field can be represented by

point size × device resolution / 72 dpi

where *point* is a typographical term meaning approximately 1/72 inch.

`spExtra` A fixed-point number equal to the average number of pixels by which each space character should be widened to fill out the line. The `spExtra` field is useful when a line of characters is to be aligned with both the left and the right margins (sometimes called *full justification*).

`fgColor` The color of the "ink" that QuickDraw uses to draw with. By default, this color is black. You can use the `ForeColor` procedure, as described in the chapter "QuickDraw Drawing," to specify any color from the eight-color system to be the foreground color in a basic graphics port. This color is recorded when drawing into a QuickDraw picture (described in the chapter "Pictures" in this book)—for example, drawing a line with a red foreground color stores a red line in the picture—but this color cannot be stored in a bitmap. When running in System 7, your application should use the `GetForeColor` procedure (described in the chapter "Color QuickDraw") to determine the foreground color instead of checking the value of this field.

`bkColor` The color of the pixels in the bitmap into which QuickDraw draws. By default, this color is white. You can use the `BackColor` procedure, as described in the chapter "QuickDraw Drawing," to specify any color from the eight-color system to be the background color in a basic graphics port. This color is recorded when drawing into a QuickDraw picture, but this color cannot be stored in a bitmap. When running in System 7, your application should use the `GetBackColor` procedure (described in the chapter "Color QuickDraw") to determine the background color instead of checking the value of this field.

`colrBit`	The plane of the color picture to draw into when printing. As in the preceding two fields, this color cannot be stored in a bitmap.
`patStretch`	A value used during output to a printer to expand patterns if necessary. Your application should not change this value.
`picSave`	The state of the picture definition. If no picture is open, this field contains `NIL`; otherwise it contains a handle to information related to the picture definition. Your application shouldn't be concerned about exactly what information the handle leads to; you may, however, save the current value of this field, set the field to `NIL` to disable the picture definition, and later restore it to the saved value to resume defining the picture. Pictures are described in the chapter "Pictures" in this book.
`rgnSave`	The state of the region definition. If no region is open, this field contains `NIL`; otherwise it contains a handle to information related to the region definition. Your application shouldn't be concerned about exactly what information the handle leads to; you may, however, save the current value of this field, set the field to `NIL` to disable the region definition, and later restore it to the saved value to resume defining the region.
`polySave`	The state of the polygon definition. If no polygon is open, this field contains `NIL`; otherwise it contains a handle to information related to the polygon definition. Your application shouldn't be concerned about exactly what information the handle leads to; you may, however, save the current value of this field, set the field to `NIL` to disable the polygon definition, and later restore it to the saved value to resume defining the polygon.
`grafProcs`	An optional pointer to a special data structure that your application can store into if you want to customize QuickDraw drawing routines or use QuickDraw in other advanced, highly specialized ways. See the chapter "QuickDraw Drawing" for more information.

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

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

Routines

This section describes the routines for initializing basic (as well as Color) QuickDraw, opening and closing graphics ports, saving and restoring graphics ports, managing port rectangles and clipping regions, and manipulating points in graphics ports.

Initializing QuickDraw

Use the `InitGraf` procedure to initialize QuickDraw at the beginning of your program, before initializing any other Toolbox managers, such as the Menu Manager and Window Manager.

InitGraf

Use the `InitGraf` procedure to initialize QuickDraw.

```
PROCEDURE InitGraf (globalPtr: Ptr);
```

`globalPtr` A pointer to the global variable `thePort`, which from Pascal can be passed as `@thePort`.

DESCRIPTION

Use the `InitGraf` procedure before initializing any other Toolbox managers, such as the Menu Manager and Window Manager. The `InitGraf` procedure initializes the global variables listed in Table 2-1 (as well as some private global variables for QuickDraw's own internal use). The `InitGraf` procedure also initializes Color QuickDraw on computers with Color QuickDraw capabilities.

Table 2-1 QuickDraw global variables

Variable	Type	Initial setting
`thePort`	`GrafPtr`	`NIL`
`white`	`Pattern`	All-white pattern
`black`	`Pattern`	All-black pattern
`gray`	`Pattern`	50% gray pattern
`ltGray`	`Pattern`	25% gray pattern
`dkGray`	`Pattern`	75% gray pattern
`arrow`	`Cursor`	Standard arrow cursor
`screenBits`	`BitMap`	Entire main screen
`randSeed`	`LongInt`	1

ASSEMBLY-LANGUAGE INFORMATION

The QuickDraw global variables are stored in reverse order, from high to low memory as listed in Table 2-1, and require the number of bytes specified by the global constant `grafSize`. Most development systems preallocate space for these global variables immediately below the location pointed to by register A5. Since `thePort` is 4 bytes, you would pass the `globalPtr` parameter as follows:

```
PEA   -4(A5)
_InitGraf
```

The `InitGraf` procedure stores this pointer to `thePort` in the location pointed to by A5.

This value is used as a base address when accessing the other QuickDraw global variables, which are accessed using negative offsets (the offsets have the same names as the Pascal global variables). For example:

```
MOVE.L   (A5),A0           ;point to first QuickDraw global
MOVE.L   randSeed(A0),A1   ;get global variable randSeed
```

SPECIAL CONSIDERATIONS

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

SEE ALSO

Listing 2-1 on page 2-16 illustrates the use of `InitGraf`.

To initialize the cursor, call the `InitCursor` procedure, which is described in the chapter "Cursor Utilities."

Opening and Closing Basic Graphics Ports

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

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

window record containing the graphics port by calling the Memory Manager procedure `DisposePtr`.

OpenPort

The `OpenPort` procedure allocates space for and initializes a basic graphics port. The Window Manager calls `OpenPort` for each black-and-white window it creates, and the `NewGWorld` procedure calls `OpenPort` for every offscreen graphics world containing a basic graphics port that it creates.

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

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

DESCRIPTION

The `OpenPort` procedure allocates space for visible and clipping regions for the graphics port specified in the `port` parameter, initializes the fields of the port's `GrafPort` record as indicated in Table 2-2, and makes that graphics port the current port (by calling `SetPort`). The Window Manager calls `OpenPort` when you create a black-and-white window; you normally won't call it yourself. You can create the graphics port pointer with the Memory Manager's `NewPtr` procedure.

Table 2-2 Initial values of a basic graphics port

Variable	Type	Initial setting
`device`	`Integer`	0 (the screen)
`portBits`	`BitMap`	`screenBits` (global variable for main screen)
`portRect`	`Rect`	`screenBits.bounds`
`visRgn`	`RgnHandle`	Handle to a rectangular region coincident with `screenBits.bounds`
`clipRgn`	`RgnHandle`	Handle to the rectangular region (–32768,–32768,32767,32767)
`bkPat`	`Pattern`	White
`fillPat`	`Pattern`	Black
`pnLoc`	`Point`	(0,0)
`pnSize`	`Point`	(1,1)
`pnMode`	`Integer`	`patCopy` pattern mode
`pnPat`	`Pattern`	Black
`pnVis`	`Integer`	0 (visible)

Table 2-2 Initial values of a basic graphics port (continued)

Variable	Type	Initial setting
txFont	Integer	0 (system font)
txFace	Style	Plain
txMode	Integer	srcOr source mode
txSize	Integer	0 (system font size)
spExtra	Fixed	0
fgColor	LongInt	blackColor
bkColor	LongInt	whiteColor
colrBit	Integer	0
patStretch	Integer	0
picSave	Handle	NIL
rgnSave	Handle	NIL
polySave	Handle	NIL
grafProcs	QDProcsPtr	NIL

SPECIAL CONSIDERATIONS

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

SEE ALSO

The `GrafPort` record is described beginning on page 2-30. Listing 2-2 on page 2-17 illustrates how to use the Window Manager function `GetNewWindow` to create a basic graphics port. The `OpenCPort` procedure (described in the chapter "Color QuickDraw") creates a color graphics port.

InitPort

You should never need to use the `InitPort` procedure. The `OpenPort` procedure calls the `InitPort` procedure, which reinitializes the fields of a basic graphics port and makes it the current port.

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

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

DESCRIPTION

The `InitPort` procedure reinitializes the fields of a `GrafPort` record that was opened with the `OpenPort` procedure, and makes it the current graphics port. The `InitPort` procedure sets the values of the port's fields to those listed in the `OpenPort` procedure description. The `InitPort` procedure does not allocate space for the visible or clipping regions.

SEE ALSO

The `InitCPort` procedure (described in the chapter "Color QuickDraw") initializes a color graphics port.

ClosePort

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

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

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

DESCRIPTION

The `ClosePort` procedure releases the memory occupied by the given graphics port's `visRgn` and `clipRgn` fields. When you're completely through with a basic graphics port, you can use this procedure and then dispose of the graphics port with the Memory Manager procedure `DisposePtr` (if it was allocated with `NewPtr`). When you call the `DisposeWindow` procedure to close or dispose of a window, it calls `ClosePort` and `DisposePtr` for you. When you use the `CloseWindow` procedure, it calls `ClosePort`, but you must call `DisposePtr`.

SPECIAL CONSIDERATIONS

If `ClosePort` isn't called before a basic graphics port is disposed of, the memory used by the visible region and the clipping region will be unrecoverable.

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

SEE ALSO

The `CloseCPort` procedure (described in the chapter "Color QuickDraw") closes a color graphics port. The `DisposeGWorld` procedure is described in the chapter "Offscreen Graphics Worlds" in this book. The `DisposeWindow` and `CloseWindow` procedures are described in the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*. The `DisposePtr` procedure is described in the chapter "Memory Manager" in *Inside Macintosh: Memory*.

Saving and Restoring Graphics Ports

If your application draws into more than one graphics port (basic or color), you can use the `SetPort` procedure to set the graphics port into which you want to draw. At times you may need to preserve the current graphics port. You can do this by using the `GetPort` procedure to save the current graphics port (basic or color), using `SetPort` to set the graphics port you want to draw in, and then using `SetPort` again when you need to restore the previous graphics port.

Note

When your application runs in Color QuickDraw or uses offscreen graphics worlds, it should use the `GetGWorld` procedure instead of `GetPort`, and it should use the `SetGWorld` procedure instead of `SetPort`. These procedures save and restore the current graphics port for basic and color graphics ports as well as offscreen graphics worlds. See the chapter "Offscreen Graphics Worlds" for more information. ◆

GetPort

To save the current graphics port (basic or color), you can use the `GetPort` procedure.

```
PROCEDURE GetPort (VAR port: GrafPtr);
```

`port` A pointer to a `GrafPort` record. If the current graphics port is a color graphics port, `GetPort` coerces its `CGrafPort` record into a `GrafPort` record.

DESCRIPTION

The `GetPort` procedure returns a pointer to the current graphics port in the `port` parameter. The current graphics port is also available through the global variable `thePort`, but you may prefer to use `GetPort` for better readability of your code. For example, your program could include `GetPort(savePort)` before setting a new graphics port, followed by `SetPort(savePort)` to restore the previous port.

SEE ALSO

Listing 2-3 on page 2-18 illustrates how to use `GetPort` to save the graphics port for the active window and `SetPort` to make an inactive window the current graphics port; then how to use `SetPort` again to restore the active window as the current graphics port. The basic graphics port is described on page 2-30. The `SetPort` procedure is described next.

When your application runs in Color QuickDraw or uses offscreen graphics worlds, it should use the `GetGWorld` procedure instead of `GetPort`. The `GetGWorld` procedure saves the current graphics port for basic and color graphics ports as well as offscreen graphics worlds. See the chapter "Offscreen Graphics Worlds" for more information.

SetPort

To change the current graphics port (basic or color), you can use the `SetPort` procedure.

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

`port` A pointer to a `GrafPort` record. Typically, you pass a pointer to a `GrafPort` record that you previously saved with the `GetPort` procedure (described in the previous section).

DESCRIPTION

The `SetPort` procedure sets the current graphics port (pointed to by the global variable `thePort`) to be that specified by the `port` parameter. All QuickDraw drawing routines affect the bitmap of, and use the local coordinate system of, the current graphics port. Each graphics port has its own graphics pen and text characteristics, which remain unchanged when the graphics port isn't selected as the current graphics port.

SEE ALSO

Listing 2-3 on page 2-18 illustrates how to use `GetPort` to save the graphics port for the active window and `SetPort` to make an inactive window the current graphics port; then how to use `SetPort` again to restore the active window as the current graphics port. The basic graphics port is described on page 2-30. The `GetPort` procedure is described on page 2-41.

When your application runs in Color QuickDraw or uses offscreen graphics worlds, it should use the `SetGWorld` procedure instead of `SetPort`. The `SetGWorld` procedure restores the current graphics port for basic and color graphics ports as well as offscreen graphics worlds. See the chapter "Offscreen Graphics Worlds" for more information.

Managing Bitmaps, Port Rectangles, and Clipping Regions

You can use the `ScrollRect`, `SetOrigin`, `GetClip`, `SetClip`, and `ClipRect` procedures to assist you when scrolling and drawing into a window. The `ScrollRect` procedure scrolls the pixels of a specified portion of a basic graphics port's bitmap (or a color graphics port's pixel map). The `SetOrigin` procedure lets you shift the coordinate plane of the current graphics port (basic or color). The `ClipRect`, `GetClip`, and `SetClip` procedures let you create, save, and set clipping regions in a graphics port (basic or color).

You can convert bitmaps (or, for color graphics ports, pixel maps) to regions using the `BitMapToRegion` function.

The `PortSize` and `MovePortTo` procedures are normally called only by Window Manager routines that manipulate the port rectangle of a window. These routines are described here for completeness.

You can use the `SetPortBits` procedure to set the bitmap for the current graphics port. This procedure was created for initial versions of QuickDraw to allow you to perform drawing and calculations on a buffer other than the screen. However, instead of using `SetPortBits`, you should use the offscreen graphics capabilities described in the chapter "Offscreen Graphics Worlds" in this book.

ScrollRect

To scroll the pixels of a specified portion of a basic graphics port's bitmap (or a color graphics port's pixel map), use the `ScrollRect` procedure.

```
PROCEDURE ScrollRect (r: Rect; dh,dv: Integer;
                      updateRgn: RgnHandle);
```

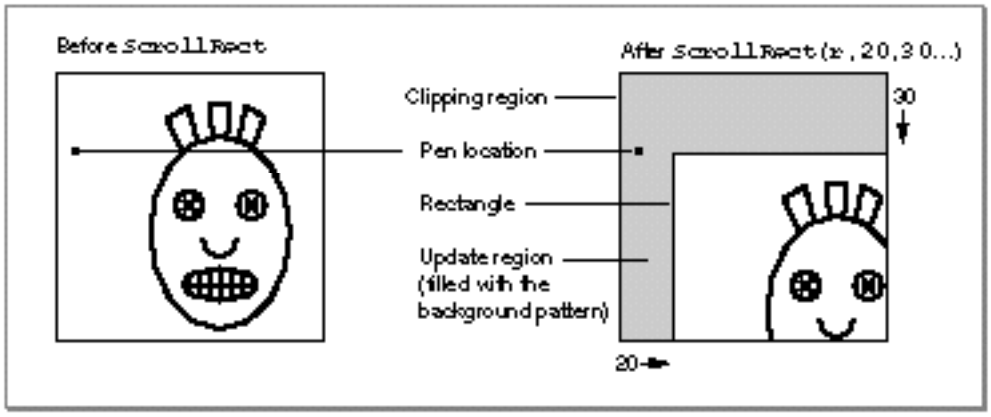
`r`	The rectangle defining the area to be scrolled.
`dh`	The horizontal distance to be scrolled.
`dv`	The vertical distance to be scrolled.
`updateRgn`	A handle to the region of the window that needs to be updated.

DESCRIPTION

The `ScrollRect` procedure shifts pixels that are inside the specified rectangle of the current graphics port. No other pixels or the bits they represent are affected. The pixels are shifted a distance of `dh` horizontally and `dv` vertically. The positive directions are to the right and down. The pixels that are shifted out of the specified rectangle are not displayed, and the bits they represent are not saved. It is up to your application to save this data.

The empty area created by the scrolling is filled with the graphics port's background pattern, and the update region is changed to this filled area, as shown in Figure 2-8.

Figure 2-8 Scrolling the image in a rectangle by using the `ScrollRect` procedure



The `ScrollRect` procedure doesn't change the local coordinate system of the graphics port; it simply moves the rectangle specified in the `r` parameter to different coordinates. Notice that `ScrollRect` doesn't move the graphics pen or the clipping region. However, because the document has moved, they're in different positions relative to the document.

By creating an update region for the window, `ScrollRect` forces an update event. After using `ScrollRect`, your application should use its own window-updating code to draw into the update region of the window.

SPECIAL CONSIDERATIONS

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

SEE ALSO

"Scrolling the Pixels in the Port Rectangle" beginning on page 2-20 provides a general discussion of the use of `ScrollRect`, and Listing 2-5 on page 2-22 illustrates how to use `ScrollRect` to scroll through a document in a window.

SetOrigin

To change the coordinates of the window origin of the port rectangle of the current graphics port (basic or color), use the `SetOrigin` procedure.

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

h	The horizontal coordinate of the upper-left corner of the port rectangle.
v	The vertical coordinate of the upper-left corner of the port rectangle.

DESCRIPTION

The `SetOrigin` procedure changes the coordinates of the upper-left corner of the current graphics port's port rectangle to the values supplied by the `h` and `v` parameters. All other points in the current graphics port's local coordinate system are calculated from this point. All subsequent drawing and calculation routines use the new coordinate system.

The `SetOrigin` procedure does not affect the screen; it does, however, affect where subsequent drawing inside the graphics port appears. The `SetOrigin` procedure does not offset the coordinates of the clipping region or the graphics pen, which therefore change position on the screen (unlike the boundary rectangle, port rectangle, and visible region, which don't change position onscreen).

Because `SetOrigin` does not move the window's clipping region, use the `GetClip` procedure to store your clipping region immediately after your first call to `SetOrigin`—if you use clipping regions in your windows. Before calling your own window-drawing routine, use the `ClipRect` procedure to define a new clipping region—to avoid drawing over your scroll bars, for example. After calling your own window-drawing routine, use the `SetClip` procedure to restore the original clipping region. You can then call `SetOrigin` again to restore the window origin to a horizontal coordinate of 0 and a vertical coordinate of 0 with your original clipping region intact.

All other routines in the Macintosh Toolbox and Operating System preserve the local coordinate system of the current graphics port. The `SetOrigin` procedure is useful for readjusting the coordinate system after a scrolling operation.

Note

The Window Manager and Control Manager always assume the window's upper-left point has a horizontal coordinate of 0 and a vertical coordinate of 0 when they draw in a window. Therefore, if you use `SetOrigin` to change the window origin, be sure to use `SetOrigin` again to return the window origin to a horizontal coordinate of 0 and a vertical coordinate of 0 before using any Window Manager or Control Manager routines. ◆

SEE ALSO

"Scrolling the Pixels in the Port Rectangle" beginning on page 2-20 provides a general discussion of the use of `SetOrigin`, and Listing 2-5 on page 2-22 illustrates how to use `SetOrigin` when scrolling through a document in a window.

PortSize

The `PortSize` procedure is normally called only by the Window Manager; it changes the size of the port rectangle of the current graphics port (basic or color).

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

`width` The width of the reset port rectangle.

`height` The height of the reset port rectangle.

DESCRIPTION

The `PortSize` procedure changes the size of the current graphics port's port rectangle. The upper-left corner of the port rectangle remains at its same location; the width and height of the port rectangle are set to the given `width` and `height`. In other words, `PortSize` moves the lower-right corner of the port rectangle to a position relative to the upper-left corner.

The `PortSize` procedure doesn't change the clipping or visible region of the graphics port, nor does it affect the local coordinate system of the graphics port; it changes only the width and height of the port rectangle. Remember that all drawing occurs only in the intersection of the boundary rectangle and the port rectangle, after being cropped to the visible region and the clipping region.

MovePortTo

The `MovePortTo` procedure is normally called only by the Window Manager; it changes the position of the port rectangle of the current graphics port (basic or color).

```
PROCEDURE MovePortTo (leftGlobal,topGlobal: Integer);
```

`leftGlobal` The horizontal distance to move the port rectangle.

`topGlobal` The vertical distance to move the port rectangle.

DESCRIPTION

The `MovePortTo` procedure changes the position of the current graphics port's port rectangle: the `leftGlobal` and `topGlobal` parameters set the distance between the upper-left corner of the boundary rectangle and the upper-left corner of the new port rectangle.

This does not affect the screen; it merely changes the location at which subsequent drawing inside the graphics port appears. Like the `PortSize` procedure, `MovePortTo` doesn't change the clipping or visible region, nor does it affect the local coordinate system of the graphics port.

GetClip

To save the clipping region of the current graphics port (basic or color), use the `GetClip` procedure.

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

`rgn` A handle to the region to be clipped to match the clipping region of the current graphics port.

DESCRIPTION

The `GetClip` procedure changes the region specified in the `rgn` parameter to one that's equivalent to the clipping region of the current graphics port. The `GetClip` procedure doesn't change the region handle.

You can use the `GetClip` and `SetClip` procedures to preserve the current clipping region: use `GetClip` to save the current port's clipping region, and use `SetClip` to restore it. If, for example, you want to draw a half-circle on the screen, you can set the clipping region to half of the square that would enclose the whole circle, and then draw the whole circle. Only the half within the clipping region is actually drawn in the graphics port.

SPECIAL CONSIDERATIONS

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

SetClip

To change the clipping region of the current graphics port (basic or color) to a region you specify, use the `SetClip` procedure.

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

`rgn` A handle to the region to be set as the current port's clipping region.

DESCRIPTION

The `SetClip` procedure changes the clipping region of the current graphics port to the region specified in the `rgn` parameter. The `SetClip` procedure doesn't change the region handle, but instead affects the clipping region itself. Since `SetClip` copies the specified region into the current graphics port's clipping region, any subsequent changes you make to the region specified in the `rgn` parameter do not affect the clipping region of the graphics port.

The initial clipping region of a graphics port is an arbitrarily large rectangle. You can set the clipping region to any arbitrary region, to aid you in drawing inside the graphics port—for example, to avoid drawing over scroll bars when drawing into a window, you could define a clipping region that excludes the scroll bars.

You can use the `GetClip` and `SetClip` procedures to preserve the current clipping region: use `GetClip` to save the current port's clipping region, and use `SetClip` to restore it.

All other system software routines preserve the current clipping region.

SPECIAL CONSIDERATIONS

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

SEE ALSO

Figure 2-4 on page 2-12 illustrates a clipping region that has been set to exclude the scroll bars of a window.

ClipRect

To change the clipping region of the current graphics port (basic or color), use the `ClipRect` procedure.

```
PROCEDURE ClipRect (r: rect);
```

`r` A rectangle to define the boundary of the new clipping region for the current graphics port.

DESCRIPTION

The `ClipRect` procedure changes the clipping region of the current graphics port to a region that's equivalent to the rectangle specified in the `r` parameter. `ClipRect` doesn't change the region handle, but it affects the clipping region itself. Since `ClipRect` makes a copy of the given rectangle, any subsequent changes you make to that rectangle do not affect the clipping region of the port.

SPECIAL CONSIDERATIONS

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

SEE ALSO

Figure 2-4 on page 2-12 illustrates a clipping region that has been set to exclude the scroll bars of a window.

BitMapToRegion

You can use the `BitMapToRegion` function to convert a bitmap or pixel map to a region.

```
FUNCTION BitMapToRegion (region: RgnHandle; bMap: BitMap): OSErr;
```

`region` A handle to a region to hold the converted `BitMap` or `PixMap` record.

`bMap` A `BitMap` or `PixMap` record.

DESCRIPTION

The `BitMapToRegion` function converts a given `BitMap` or `PixMap` record to a region. You would generally use this region later for drawing operations. The `region` parameter must be a valid region handle created with the `NewRgn` function (described in the chapter "QuickDraw Drawing"). The old region contents are lost.

The `bMap` parameter may be either a `BitMap` or `PixMap` record. If you pass a `PixMap` record, its pixel depth must be 1.

RESULT CODES

`pixmapTooDeepErr`	–148	Pixel map is deeper than 1 bit per pixel
`rgnTooBigErr`	–500	Bitmap would convert to a region greater than 64 KB

SetPortBits

Although you should never need to do so, you can set the bitmap for the current basic graphics port by using the `SetPortBits` procedure.

```
PROCEDURE SetPortBits (bm: BitMap);
```

`bm` A `BitMap` record.

DESCRIPTION

The `SetPortBits` procedure sets the `portBits` field of the current graphics port to any previously defined bitmap. Be sure to prepare all fields of the `BitMap` record before you call `SetPortBits`.

SPECIAL CONSIDERATIONS

The `SetPortBits` procedure, created in early versions of QuickDraw, allows you to perform all normal drawing and calculations on a buffer other than the screen—for example, copying a small offscreen image onto the screen with the `CopyBits` procedure. However, instead of using `SetPortBits`, you should use the more powerful offscreen graphics capabilities described in the chapter "Offscreen Graphics Worlds."

Manipulating Points in Graphics Ports

Each graphics port (basic or color) has its own local coordinate system. Some Toolbox routines return or expect points that are expressed in the global coordinate system, while others use local coordinates. For example, when the Event Manager function `WaitNextEvent` reports an event, it gives the cursor location (also called the *mouse location*) in global coordinates; but when you call the Control Manager function `FindControl` to find out whether the user clicked a control in one of your windows, you pass the cursor location in local coordinates. You can use the `GlobalToLocal` procedure to convert global coordinates to local coordinates, and you can use the `LocalToGlobal` procedure for the reverse.

You can also use the `SetPt` procedure to create a point, the `EqualPt` function to compare two points, and the `AddPt` procedure, `SubPt` procedure, and `DeltaPoint` function to shift points. To determine whether the pixel associated with a point is black or white, use the `GetPixel` function.

GlobalToLocal

To convert the coordinates of a point from global coordinates to the local coordinates of the current graphics port (basic or color), use the `GlobalToLocal` procedure.

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

`pt` The point whose global coordinates are to be converted to local coordinates.

DESCRIPTION

The `GlobalToLocal` procedure takes a point expressed in global coordinates (where the upper-left corner of the main screen has coordinates [0,0]) and converts it into the local coordinates of the current graphics port.

SEE ALSO

Listing 2-4 on page 2-19 illustrates how to use `GlobalToLocal` to convert a point in an event reported by the Event Manager function `WaitNextEvent` to local coordinates as required by the Control Manager function `FindControl`.

LocalToGlobal

To convert a point's coordinates from the local coordinates of the current graphics port (basic or color) to global coordinates, use the `LocalToGlobal` procedure.

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

`pt` The point whose local coordinates are to be converted to global coordinates.

DESCRIPTION

The `LocalToGlobal` procedure converts the given point from the current graphics port's local coordinate system into the global coordinate system (where the upper-left corner of the main screen has coordinates [0,0]). This global point can then be compared to other global points, or it can be changed into the local coordinates of another graphics port.

Because a rectangle is defined by two points, you can convert a rectangle into global coordinates with two calls to `LocalToGlobal`. In conjunction with `LocalToGlobal`, you can use the `OffsetRect`, `OffsetRgn`, or `OffsetPoly` procedures (which are described in the chapter "QuickDraw Drawing") to convert a rectangle, region, or polygon into global coordinates.

AddPt

To add the coordinates of two points, use the `AddPt` procedure.

```
PROCEDURE AddPt (srcPt: Point; VAR dstPt: Point);
```

`srcPt` A point, the coordinates of which are to be added to the point in the `dstPt` parameter.

`dstPt` On input: a point, the coordinates of which are to be added to the point in the `srcPt` parameter. Upon completion: the result of adding the coordinates of the points in the `srcPt` and `dstPt` parameters.

DESCRIPTION

The `AddPt` procedure adds the coordinates of the point specified in the `srcPt` parameter to the coordinates of the point specified in the `dstPt` parameter, and returns the result in the `dstPt` parameter.

SubPt

To subtract the coordinates of one point from another, you can use the `SubPt` procedure.

```
PROCEDURE SubPt (srcPt: Point; VAR dstPt: Point);
```

`srcPt` A point, the coordinates of which are to be subtracted from those specified in the `dstPt` parameter.

`dstPt` On input: a point, from whose coordinates are to be subtracted those specified in the `srcPt` parameter. Upon completion: the result of subtracting the coordinates of the points in the `srcPt` parameter from the coordinates of the points in the `dstPt` parameter.

DESCRIPTION

The `SubPt` procedure subtracts the coordinates of the point specified in the `srcPt` parameter from the coordinates of the point specified in the `dstPt` parameter, and returns the result in the `dstPt` parameter.

To get the results of coordinate subtraction returned as a function result, you can instead use the `DeltaPoint` function. Note, however, that the parameters in these two routines are reversed.

DeltaPoint

To subtract the coordinates of one point from another, you can use the `DeltaPoint` function.

```
FUNCTION DeltaPoint (ptA: Point; ptB: Point): LongInt;
```

`ptA` A point, from whose coordinates are to be subtracted those specified in the `ptB` parameter.

`ptB` A point, the coordinates of which are to be subtracted from those specified in the `ptA` parameter.

DESCRIPTION

The `DeltaPoint` function subtracts the coordinates of the point specified in the `ptB` parameter from the coordinates of the point specified in the `ptA` parameter, and returns the result as its function result.

To get the results of coordinate subtraction, you can instead use the `SubPt` procedure. Note, however, that the parameters in these two routines are reversed.

SetPt

To assign two coordinates to a point, use the `SetPt` procedure.

```
PROCEDURE SetPt (VAR pt: Point; h,v: Integer);
```

`pt`	The point to be given new coordinates.
`h`	The horizontal value of the new coordinates.
`v`	The vertical value of the new coordinates.

DESCRIPTION

The `SetPt` procedure assigns the horizontal coordinate specified in the `h` parameter and the vertical coordinate specified in the `v` parameter to the point returned in the `pt` parameter.

EqualPt

To determine whether the coordinates of two given points are equal, use the `EqualPt` function.

```
FUNCTION EqualPt (pt1,pt2: Point): Boolean;
```

`pt1,pt2`	The two points to be compared.

DESCRIPTION

The `EqualPt` function compares the points specified in the `pt1` and `pt2` parameters and returns `TRUE` if their coordinates are equal or `FALSE` if they are not.

GetPixel

To determine whether the pixel associated with a point is black or white, use the `GetPixel` function.

```
FUNCTION GetPixel (h,v: Integer): Boolean;
```

`h`	The horizontal coordinate of the point for the pixel to be tested.
`v`	The vertical coordinate of the point for the pixel to be tested.

DESCRIPTION

The `GetPixel` function examines the pixel at the point specified by the `h` and `v` parameters and returns `TRUE` if the pixel is black or `FALSE` if it is `white`.

The selected pixel is immediately below and to the right of the point whose coordinates you supply in the `h` and `v` parameters, in the local coordinates of the current graphics port. There's no guarantee that the specified pixel actually belongs to the current graphics port, however; it may have been drawn in a graphics port overlapping the current one. To see if the point indeed belongs to the current graphics port, you could use the `PtInRgn` function (described in the chapter "QuickDraw Drawing" in this book) to test whether the point is in the current graphics port's visible region, as shown here.

```
PtInRgn(pt, thePort^.visRgn);
```

Summary of Basic QuickDraw

Pascal Summary

Data Types

```
TYPE  Point =
      RECORD CASE Integer OF
         0: (v:         Integer;    {vertical coordinate}
             h:         Integer);   {horizontal coordinate}
         1: (vh:        ARRAY[VHSelect] OF Integer);
      END;

      Rect =
      RECORD CASE Integer OF        {cases: 4 boundaries or 2 corners}
         0: (top:       Integer;    {upper boundary of rectangle}
             left:      Integer;    {left boundary of rectangle}
             bottom:    Integer;    {lower boundary of rectangle}
             right:     Integer);   {right boundary of rectangle}
         1: (topLeft:   Point;      {upper-left corner of rectangle}
             botRight:  Point);     {lower-right corner of rectangle}
      END;

      RgnHandle = ^RgnPtr;
      RgnPtr =    ^Region;
      Region =
      RECORD
         rgnSize: Integer; {size in bytes}
         rgnBBox: Rect;    {enclosing rectangle}
         {more data if not rectangular}
      END;

      BitMap =
      RECORD
         baseAddr:   Ptr;     {pointer to bit image}
         rowBytes:   Integer; {row width}
         bounds:     Rect;    {boundary rectangle}
      END;
```

```
GrafPtr =      ^GrafPort;
WindowPtr =    GrafPtr;

GrafPort =  {basic graphics port}
RECORD
   device:     Integer;    {device-specific information}
   portBits:   BitMap;     {bitmap}
   portRect:   Rect;       {port rectangle}
   visRgn:     RgnHandle;  {visible region}
   clipRgn:    RgnHandle;  {clipping region}
   bkPat:      Pattern;    {background pattern}
   fillPat:    Pattern;    {fill pattern}
   pnLoc:      Point;      {pen location}
   pnSize:     Point;      {pen size}
   pnMode:     Integer;    {pattern mode}
   pnPat:      Pattern;    {pen pattern}
   pnVis:      Integer;    {pen visibility}
   txFont:     Integer;    {font number for text}
   txFace:     Style;      {text's font style}
   txMode:     Integer;    {source mode for text}
   txSize:     Integer;    {font size for text}
   spExtra:    Fixed;      {extra space}
   fgColor:    LongInt;    {foreground color}
   bkColor:    LongInt;    {background color}
   colrBit:    Integer;    {color bit}
   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:  QDProcsPtr; {low-level drawing routines}
END;
```


Routines

Initializing QuickDraw

```
PROCEDURE InitGraf          (globalPtr: Ptr);
```

Opening and Closing Basic Graphics Ports

```
PROCEDURE OpenPort          (port: GrafPtr);
PROCEDURE InitPort          (port: GrafPtr);
PROCEDURE ClosePort         (port: GrafPtr);
```

Saving and Restoring Graphics Ports

```
PROCEDURE GetPort           (VAR port: GrafPtr);
PROCEDURE SetPort           (port: GrafPtr);
```

Managing Bitmaps, Port Rectangles, and Clipping Regions

```
PROCEDURE ScrollRect        (r: Rect; dh,dv: Integer; updateRgn: RgnHandle);
PROCEDURE SetOrigin         (h,v: Integer);
PROCEDURE PortSize          (width,height: Integer);
PROCEDURE MovePortTo        (leftGlobal,topGlobal: Integer);
PROCEDURE GetClip           (rgn: RgnHandle);
PROCEDURE SetClip           (rgn: RgnHandle);
PROCEDURE ClipRect          (r: Rect);
FUNCTION BitMapToRegion     (region: RgnHandle; bMap: BitMap): OSErr;
PROCEDURE SetPortBits       (bm: BitMap);
```

Manipulating Points in Graphics Ports

```
PROCEDURE GlobalToLocal     (VAR pt: Point);
PROCEDURE LocalToGlobal     (VAR pt: Point);
PROCEDURE AddPt             (srcPt: Point; VAR dstPt: Point);
PROCEDURE SubPt             (srcPt: Point; VAR dstPt: Point);
FUNCTION DeltaPoint         (ptA: Point; ptB: Point): LongInt;
PROCEDURE SetPt             (VAR pt: Point; h,v: Integer);
FUNCTION EqualPt            (pt1,pt2: Point): Boolean;
FUNCTION GetPixel           (h,v: Integer): Boolean;
```

C Summary

Data Types

```
struct Point {
      short v;    /* vertical coordinate */
      short h;    /* horizontal coordinate */
   };
```

```
struct Rect {
      short top;     /* upper boundary of rectangle */
      short left;    /* left boundary of rectangle */
      short bottom;  /* lower boundary of rectangle */
      short right;   /* right boundary of rectangle */
   };

struct Region {
      short rgnSize;          /* size in bytes */
      Rect  rgnBBox;          /* enclosing rectangle */
      /* more data if not rectangular */
   };

typedef struct Region Region;
typedef Region *RgnPtr, **RgnHandle;

struct BitMap {
      Ptr   baseAddr;         /* pointer to bit image */
      short rowBytes;         /* row width */
      Rect  bounds;           /* boundary rectangle */
   };

struct GrafPort {             /* basic graphics port */
      short       device;     /* device-specific information */
      BitMap      portBits;   /* bitmap */
      Rect        portRect;   /* port rectangle */
      RgnHandle   visRgn;     /* visible region */
      RgnHandle   clipRgn;    /* clipping region */
      Pattern     bkPat;      /* background pattern */
      Pattern     fillPat;    /* fill pattern */
      Point       pnLoc;      /* pen location */
      Point       pnSize;     /* pen size */
      short       pnMode;     /* pattern mode */
      Pattern     pnPat;      /* pen pattern */
      short       pnVis;      /* pen visibility */
      short       txFont;     /* font number for text */
      Style       txFace;     /* text's font style */
      char        filler;
      short       txMode;     /* source mode for text */
      short       txSize;     /* font size for text */
      Fixed       spExtra;    /* extra space */
      long        fgColor;    /* foreground color */
      long        bkColor;    /* background color */
      short       colrBit;    /* color bit */
```

```
    short       patStretch; /* used internally */
    Handle      picSave;    /* picture being saved, used internally */
    Handle      rgnSave;    /* region being saved, used internally */
    Handle      polySave;   /* polygon being saved, used internally */
    QDProcsPtr  grafProcs;  /* low-level drawing routines */
};

typedef struct GrafPort GrafPort;
typedef GrafPort *GrafPtr;
typedef GrafPtr WindowPtr;
```

Functions

Initializing QuickDraw

```
pascal void InitGraf        (void *globalPtr);
```

Opening and Closing Basic Graphics Ports

```
pascal void OpenPort        (GrafPtr port);
pascal void InitPort        (GrafPtr port);
pascal void ClosePort       (GrafPtr port);
```

Saving and Restoring Graphics Ports

```
pascal void GetPort         (GrafPtr *port);
pascal void SetPort         (GrafPtr port);
```

Managing Bitmaps, Port Rectangles, and Clipping Regions

```
pascal void ScrollRect      (const Rect *r, short dh, short dv,
                             RgnHandle updateRgn);
pascal void SetOrigin       (short h, short v);
pascal void PortSize        (short width, short height);
pascal void MovePortTo      (short leftGlobal, short topGlobal);
pascal void GetClip         (RgnHandle rgn);
pascal void SetClip         (RgnHandle rgn);
pascal void ClipRect        (const Rect *r);
pascal OSErr BitMapToRegion (RgnHandle region, const BitMap *bMap);
pascal void SetPortBits     (const BitMap *bm);
```

Manipulating Points in Graphics Ports

```
pascal void GlobalToLocal   (Point *pt);
pascal void LocalToGlobal   (Point *pt);
pascal void AddPt           (Point src, Point *dst);
pascal void SubPt           (Point src, Point *dst);
pascal long DeltaPoint      (Point ptA, Point ptB);
pascal void SetPt           (Point *pt, short h, short v);
pascal Boolean EqualPt      (Point pt1, Point pt2);
pascal Boolean GetPixel     (short h, short v);
```


Assembly-Language Summary

Data Structures

Point Data Structure

0	v	word	vertical coordinate
2	h	word	horizontal coordinate

Rectangle Data Structure

0	topLeft	long	upper-left corner of rectangle
4	botRight	long	lower-right corner of rectangle
0	top	word	upper boundary of rectangle
2	left	word	left boundary of rectangle
4	bottom	word	lower boundary of rectangle
6	right	word	right boundary of rectangle

Region Data Structure

0	rgnSize	word	size in bytes
2	rgnBBox	8 bytes	enclosing rectangle
10	rgnData	array	region data

Bitmap Data Structure

0	baseAddr	long	pointer to bit image
4	rowBytes	word	row width
6	bounds	8 bytes	boundary rectangle

GrafPort Data Structure

0	device	word	device-specific information
2	portBits	14 bytes	bitmap
16	portBounds	8 bytes	boundary rectangle
24	portRect	8 bytes	port rectangle
32	visRgn	long	visible region
36	clipRgn	long	clipping region
40	bkPat	8 bytes	background pattern
48	fillPat	8 bytes	fill pattern
56	pnLoc	long	pen location
60	pnSize	long	pen size
64	pnMode	word	pattern mode
66	pnPat	8 bytes	pen pattern
74	pnVis	word	pen visibility
76	txFont	word	font number for text
78	txFace	word	text's font style
80	txMode	word	source mode for text
82	txSize	word	font size for text
84	spExtra	long	extra space
88	fgColor	long	foreground color
92	bkColor	long	background color
96	colrBit	word	color bit
98	patStretch	word	used internally
100	picSave	long	picture being saved, used internally
104	rgnSave	long	region being saved, used internally
108	polySave	long	polygon being saved, used internally
112	grafProcs	long	low-level drawing routines

Global Variables

arrow	The standard arrow cursor.
black	An all-black pattern.
dkGray	A 75% gray pattern.
gray	A 50% gray pattern.
ltGray	A 25% gray pattern.
randSeed	Where the random sequence begins.
screenBits	The main screen.
thePort	The current graphics port.
white	An all-white pattern.

Result Codes

pixmapTooDeepErr	–148	Pixel map is deeper than 1 bit per pixel
rgnTooBigErr	–500	Bitmap would convert to a region greater than 64 KB