

INSIDE MACINTOSH

Imaging With QuickDraw

Apple Computer, Inc.
© 1994 Apple Computer, Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc. Printed in the United States of America.

No licenses, express or implied, are granted with respect to any of the technology described in this book. Apple retains all intellectual property rights associated with the technology described in this book. This book is intended to assist application developers to develop applications only for Apple Macintosh computers.

Every effort has been made to ensure that the information in this manual is accurate. Apple is not responsible for printing or clerical errors.

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, APDA, AppleLink, AppleTalk, ImageWriter, LaserWriter, Macintosh, MPW, PowerBook, and StyleWriter are trademarks of Apple Computer, Inc., registered in the United States and other countries.

ColorSync, Finder, Geneva, QuickDraw, QuickTime, ResEdit, System 7, and TrueType are trademarks of Apple Computer, Inc.

Adobe Illustrator, Adobe Photoshop, and PostScript are trademarks of Adobe Systems Incorporated, which may be registered in certain jurisdictions.

America Online is a service mark of Quantum Computer Services, Inc.

Classic is a registered trademark licensed to Apple Computer, Inc.

CompuServe is a registered service mark of CompuServe, Inc.

FrameMaker is a registered trademark of Frame Technology Corporation.

Helvetica and Palatino are registered trademarks of Linotype Company.

Internet is a trademark of Digital Equipment Corporation.

ITC Zapf Dingbats is a registered trademark of International Typeface Corporation.

MacPaint is a registered trademark of Claris Corporation.

NuBus is a trademark of Texas Instruments.

Motorola is a registered trademark of Motorola Corporation.

Optrotech is a trademark of Orbotech Corporation.

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Figures, Tables, and Listings xiii

Preface About This Book xxi

Format of a Typical Chapter xxii
Conventions Used in This Book xxiii
Special Fonts xxiii
Types of Notes xxiii
Empty Strings xxiv
Assembly-Language Information xxiv
The Development Environment xxiv

Chapter 1 Introduction to QuickDraw 1-1

Drawing Environments 1-4
QuickDraw's Coordinate Plane 1-6
Images 1-10
Colors 1-17
Indexed Colors 1-19
Direct Colors 1-20
Multiple Screens 1-21
From Memory Bits to Onscreen Pixels 1-24
From Memory Bits to Printers 1-26
Other Graphics Managers 1-28

Chapter 2 Basic QuickDraw 2-1

About Basic QuickDraw 2-3
The Mathematical Foundations of QuickDraw 2-4
The Coordinate Plane 2-4
Points 2-4
Rectangles 2-5
Regions 2-7
The Black-and-White Drawing Environment: Basic Graphics Ports 2-7
Bitmaps 2-9
The Graphics Port Drawing Area 2-11
Graphics Port Bit Patterns 2-13
The Graphics Pen 2-13
Text in a Graphics Port 2-13

The Limited Colors of a Basic Graphics Port 2-14
Other Fields 2-14
Using Basic QuickDraw 2-14
Initializing Basic QuickDraw 2-16
Creating Basic Graphics Ports 2-16
Setting the Graphics Port 2-18
Switching Between Global and Local Coordinate Systems 2-19
Scrolling the Pixels in the Port Rectangle 2-20
Basic QuickDraw Reference 2-26
Data Structures 2-26
Routines 2-36
Initializing QuickDraw 2-36
Opening and Closing Basic Graphics Ports 2-37
Saving and Restoring Graphics Ports 2-41
Managing Bitmaps, Port Rectangles, and Clipping Regions 2-43
Manipulating Points in Graphics Ports 2-51
Summary of Basic QuickDraw 2-56
Pascal Summary 2-56
Data Types 2-56
Routines 2-57
C Summary 2-58
Data Types 2-58
Functions 2-60
Assembly-Language Summary 2-61
Data Structures 2-61
Global Variables 2-62
Result Codes 2-62

Chapter 3 QuickDraw Drawing 3-1

About QuickDraw Drawing 3-3
The Graphics Pen 3-4
Bit Patterns 3-5
Boolean Transfer Modes With 1-Bit Pixels 3-8
Lines and Shapes 3-11
Defining Lines and Shapes 3-11
Framing Shapes 3-12
Painting and Filling Shapes 3-12
Erasing Shapes 3-12
Inverting Shapes 3-13
Other Graphic Entities 3-13
The Eight Basic QuickDraw Colors 3-14
Drawing With QuickDraw 3-16
Drawing Lines 3-17
Drawing Rectangles 3-22

Drawing Ovals, Arcs, and Wedges 3-25
Drawing Regions and Polygons 3-27
Performing Calculations and Other Manipulations of Shapes 3-31
Copying Bits Between Graphics Ports 3-32
Customizing QuickDraw's Low-Level Routines 3-35
QuickDraw Drawing Reference 3-36
Data Structures 3-36
Routines 3-41
Managing the Graphics Pen 3-41
Changing the Background Bit Pattern 3-48
Drawing Lines 3-49
Creating and Managing Rectangles 3-52
Drawing Rectangles 3-58
Drawing Rounded Rectangles 3-63
Drawing Ovals 3-68
Drawing Arcs and Wedges 3-71
Creating and Managing Polygons 3-78
Drawing Polygons 3-81
Creating and Managing Regions 3-85
Drawing Regions 3-100
Scaling and Mapping Points, Rectangles, Polygons, and Regions 3-104
Calculating Black-and-White Fills 3-108
Copying Images 3-112
Drawing With the Eight-Color System 3-122
Determining Whether QuickDraw Has Finished Drawing 3-125
Getting Pattern Resources 3-126
Customizing QuickDraw Operations 3-129
Resources 3-140
The Pattern Resource 3-140
The Pattern List Resource 3-141
Summary of QuickDraw Drawing 3-142
Pascal Summary 3-142
Constants 3-142
Data Types 3-144
Routines 3-145
C Summary 3-149
Constants 3-149
Data Types 3-151
Functions 3-152
Assembly-Language Summary 3-157
Data Structures 3-157
Global Variables 3-158

Chapter 4 Color QuickDraw 4-1

About Color QuickDraw 4-4
RGB Colors 4-4
The Color Drawing Environment: Color Graphics Ports 4-5
Pixel Maps 4-9
Pixel Patterns 4-12
Color QuickDraw's Translation of RGB Colors to Pixel Values 4-13
Colors on Grayscale Screens 4-17
Using Color QuickDraw 4-18
Initializing Color QuickDraw 4-19
Creating Color Graphics Ports 4-20
Drawing With Different Foreground Colors 4-21
Drawing With Pixel Patterns 4-23
Copying Pixels Between Color Graphics Ports 4-26
Boolean Transfer Modes With Color Pixels 4-32
Dithering 4-37
Arithmetic Transfer Modes 4-38
Highlighting 4-41
Color QuickDraw Reference 4-44
Data Structures 4-45
Color QuickDraw Routines 4-63
Opening and Closing Color Graphics Ports 4-63
Managing a Color Graphics Pen 4-67
Changing the Background Pixel Pattern 4-68
Drawing With Color QuickDraw Colors 4-70
Determining Current Colors and Best Intermediate Colors 4-79
Calculating Color Fills 4-82
Creating, Setting, and Disposing of Pixel Maps 4-85
Creating and Disposing of Pixel Patterns 4-87
Creating and Disposing of Color Tables 4-91
Retrieving Color QuickDraw Result Codes 4-94
Customizing Color QuickDraw Operations 4-96
Reporting Data Structure Changes to QuickDraw 4-97
Application-Defined Routine 4-101
Resources 4-102
The Pixel Pattern Resource 4-103
The Color Table Resource 4-104
The Color Icon Resource 4-105
Summary of Color QuickDraw 4-107
Pascal Summary 4-107
Constants 4-107
Data Types 4-109
Color QuickDraw Routines 4-113
Application-Defined Routine 4-115

C Summary 4-115
Constants 4-115
Data Types 4-118
Color QuickDraw Functions 4-122
Application-Defined Function 4-124
Assembly-Language Summary 4-124
Data Structures 4-124
Result Codes 4-128

Chapter 5 Graphics Devices 5-1

About Graphics Devices 5-3
Using Graphics Devices 5-6
Optimizing Your Images for Different Graphics Devices 5-8
Zooming Windows on Multiscreen Systems 5-9
Setting a Device's Pixel Depth 5-13
Exceptional Cases When Working With Color Devices 5-13
Graphics Devices Reference 5-14
Data Structures 5-15
Routines for Graphics Devices 5-19
Creating, Setting, and Disposing of GDevice Records 5-19
Getting the Available Graphics Devices 5-25
Determining the Characteristics of a Video Device 5-29
Changing the Pixel Depth for a Video Device 5-33
Application-Defined Routine 5-35
Resource 5-37
The Screen Resource 5-37
Summary of Graphics Devices 5-38
Pascal Summary 5-38
Constants 5-38
Data Types 5-39
Routines for Graphics Devices 5-40
Application-Defined Routine 5-40
C Summary 5-41
Constants 5-41
Data Types 5-41
Functions for Graphics Devices 5-43
Application-Defined Function 5-44
Assembly-Language Summary 5-44
Data Structure 5-44
Global Variables 5-44

Chapter 6 Offscreen Graphics Worlds 6-1

About Offscreen Graphics Worlds 6-3
Using Offscreen Graphics Worlds 6-4
Creating an Offscreen Graphics World 6-5
Setting the Graphics Port for an Offscreen Graphics World 6-8
Drawing Into an Offscreen Graphics World 6-8
Copying an Offscreen Image Into a Window 6-9
Updating an Offscreen Graphics World 6-9
Creating a Mask and a Source Image in Offscreen Graphics Worlds 6-10
Offscreen Graphics Worlds Reference 6-12
Data Structures 6-12
Routines 6-16
Creating, Altering, and Disposing of Offscreen Graphics Worlds 6-16
Saving and Restoring Graphics Ports and Offscreen Graphics Worlds 6-27
Managing an Offscreen Graphics World's Pixel Image 6-30
Summary of Offscreen Graphics Worlds 6-40
Pascal Summary 6-40
Constants 6-40
Data Types 6-41
Routines 6-42
C Summary 6-43
Constants 6-43
Data Types 6-44
Functions 6-45
Assembly-Language Summary 6-46
Result Codes 6-46

Chapter 7 Pictures 7-1

About Pictures 7-4
Picture Formats 7-5
Opcodes: Drawing Commands and Picture Comments 7-6
Color Pictures in Basic Graphics Ports 7-6
'PICT' Files, 'PICT' Resources, and the 'PICT' Scrap Format 7-7
The Picture Utilities 7-8
Using Pictures 7-8
Creating and Drawing Pictures 7-10
Opening and Drawing Pictures 7-13
Drawing a Picture Stored in a 'PICT' File 7-13
Drawing a Picture Stored in the Scrap 7-17
Defining a Destination Rectangle 7-18
Drawing a Picture Stored in a 'PICT' Resource 7-20
Saving Pictures 7-21
Gathering Picture Information 7-24

Pictures Reference 7-26
Data Structures 7-27
QuickDraw and Picture Utilities Routines 7-36
Creating and Disposing of Pictures 7-36
Drawing Pictures 7-43
Collecting Picture Information 7-46
Application-Defined Routines 7-61
Resources 7-67
The Picture Resource 7-67
The Color-Picking Method Resource 7-68
Summary of Pictures and the Picture Utilities 7-69
Pascal Summary 7-69
Constants 7-69
Data Types 7-69
Routines 7-72
Application-Defined Routines 7-73
C Summary 7-73
Constants 7-73
Data Types 7-74
Functions 7-76
Application-Defined Functions 7-77
Assembly-Language Summary 7-78
Data Structures 7-78
Trap Macros 7-80
Result Codes 7-80

Chapter 8 Cursor Utilities 8-1

Errata for Chapter 8: Cursor Utilities 8-2
About the Cursor 8-3
Using the Cursor Utilities 8-5
Initializing the Cursor 8-6
Changing the Appearance of the Cursor 8-7
Creating an Animated Cursor 8-13
Cursor Utilities Reference 8-16
Data Structures 8-16
Routines 8-21
Initializing Cursors 8-21
Changing Black-and-White Cursors 8-24
Changing Color Cursors 8-25
Hiding and Showing Cursors 8-28
Displaying Animated Cursors 8-31
Resources 8-33
The Cursor Resource 8-33
The Color Cursor Resource 8-34
The Animated Cursor Resource 8-36

Summary of Cursor Utilities 8-38
Pascal Summary 8-38
Constants 8-38
Data Types 8-38
Routines 8-39
C Summary 8-40
Constants 8-40
Data Types 8-41
Functions 8-42
Assembly-Language Summary 8-43
Data Structures 8-43
Global Variables 8-43

Chapter 9 Printing Manager 9-1

About the Printing Manager 9-3
The Printing Graphics Port 9-4
Getting Printing Preferences From the User 9-5
QuickDraw and PostScript Printer Drivers 9-8
Page and Paper Rectangles 9-10
Printer Resolution 9-11
The TPrint Record and the Printing Loop 9-11
Print Status Dialog Boxes 9-13
Using the Printing Manager 9-15
Creating and Using a TPrint Record 9-17
Printing a Document 9-18
Printing From the Finder 9-25
Providing Names of Documents Being Printed 9-27
Printing Hints 9-27
Getting and Setting Printer Information 9-28
Determining and Setting the Resolution of the Current Printer 9-30
Determining Page Orientation 9-32
Enhancing Draft-Quality Printing 9-33
Altering the Style or Job Dialog Box 9-35
Writing an Idle Procedure 9-38
Handling Printing Errors 9-41
Printing Manager Reference 9-43
Data Structures 9-44
Printing Manager Routines 9-57
Opening and Closing the Printing Manager 9-57
Initializing and Validating TPrint Records 9-58
Displaying and Customizing the Print Dialog Boxes 9-61
Printing a Document 9-66
Optimizing Printing 9-72

Handling Printing Errors 9-75
Low-Level Routines 9-78
Application-Defined Routines 9-84
Summary of the Printing Manager 9-87
Pascal Summary 9-87
Constants 9-87
Data Types 9-88
Printing Manager Routines 9-92
Application-Defined Routines 9-93
C Summary 9-94
Constants 9-94
Data Types 9-95
Printing Manager Functions 9-99
Application-Defined Functions 9-101
Assembly-Language Summary 9-101
Data Structures 9-101
Trap Macros 9-103
Global Variable 9-103
Result Codes 9-104

Appendix A Picture Opcodes A-1

Version and Header Opcodes A-3
Picture Opcode Data Types A-4
Opcodes in Pictures A-5
A Sample Extended Version 2 Picture A-22
A Sample Version 2 Picture A-24
A Sample Version 1 Picture A-25

Appendix B Using Picture Comments for Printing B-1

About Picture Comments B-3
Maintaining Device Independence B-8
Synchronizing QuickDraw and PostScript Printer Drivers B-10
Using Text Picture Comments B-11
Disabling and Reenabling Line Layout B-11
Delimiting Strings B-16
Rotating Text B-17
Using Graphics Picture Comments B-22
Drawing Polygons B-23
Rotating Graphics B-29
Using Line-Drawing Picture Comments B-33
Drawing Dashed Lines B-33
Using Fractional Line Widths B-35

Using PostScript Picture Comments B-38
Calling PostScript Routines Directly B-38
Optimizing PostScript Printing B-39
Picture Comments to Avoid B-40
Including Constants and Data Types for Picture Comments B-42

Glossary GL-1

Index IN-1

Figures, Tables, and Listings

Chapter 1 Introduction to QuickDraw 1-1

Figure 1-1 A grayscale image representing bits in memory 1-6
Figure 1-2 The QuickDraw global coordinate plane 1-7
Figure 1-3 A window's local and global coordinate systems 1-8
Figure 1-4 The coordinate plane 1-8
Figure 1-5 Points and pixels 1-9
Figure 1-6 Drawing a line 1-10
Figure 1-7 Lines drawn with different bit patterns and pen sizes 1-12
Figure 1-8 A rectangle 1-12
Figure 1-9 An oval 1-13
Figure 1-10 An arc and a wedge 1-14
Figure 1-11 A rounded rectangle 1-15
Figure 1-12 A polygon 1-15
Figure 1-13 Two regions 1-16
Figure 1-14 A simple QuickDraw picture 1-16
Figure 1-15 Filling and framing various shapes 1-17
Figure 1-16 A two-screen system 1-21
Figure 1-17 The `GDevice` record and pixel map for a 4-bit video card 1-22
Figure 1-18 The indexed-pixel path 1-24
Figure 1-19 The direct-pixel path 1-25
Figure 1-20 The job dialog box for a StyleWriter printer 1-26
Figure 1-21 The job dialog box for a LaserWriter printer 1-27

Chapter 2 Basic QuickDraw 2-1

Figure 2-1 The `GrafPort` record and the `BitMap` record 2-8
Figure 2-2 A bit image 2-9
Figure 2-3 Relationship of the boundary rectangle and the port rectangle to the global coordinate system 2-10
Figure 2-4 Comparing the boundary rectangle, port rectangle, visible region, and clipping region 2-12
Figure 2-5 Moving a document relative to its window 2-21
Figure 2-6 Updating the contents of a scrolled window 2-24
Figure 2-7 Restoring the window origin of the port rectangle to a horizontal coordinate of 0 and a vertical coordinate of 0 2-25
Figure 2-8 Scrolling the image in a rectangle by using the `ScrollRect` procedure 2-44

Table 2-1 QuickDraw global variables 2-36
Table 2-2 Initial values of a basic graphics port 2-38

Listing 2-1 Initializing QuickDraw 2-16
Listing 2-2 Using the Window Manager to create a basic graphics port 2-17
Listing 2-3 Saving and restoring a graphics port 2-18
Listing 2-4 Changing global coordinates to local coordinates 2-19
Listing 2-5 Using `ScrollRect` to scroll the bits displayed in the window 2-22

Chapter 3 QuickDraw Drawing 3-1

Figure 3-1 A graphics pen 3-4
Figure 3-2 A bit pattern 3-5
Figure 3-3 Windows filled with the predefined bit patterns 3-7
Figure 3-4 Examples of Boolean transfer modes 3-10
Figure 3-5 Using the `LineTo` procedure 3-17
Figure 3-6 Drawing lines 3-18
Figure 3-7 Using the `LineTo` and `Line` procedures 3-19
Figure 3-8 Resizing the pen 3-19
Figure 3-9 Changing the pen pattern 3-21
Figure 3-10 Two ways to specify a rectangle 3-22
Figure 3-11 Drawing rectangles 3-22
Figure 3-12 Painting and filling rectangles 3-23
Figure 3-13 Drawing ovals 3-25
Figure 3-14 Drawing an arc and a wedge 3-26
Figure 3-15 A shape created by a region 3-28
Figure 3-16 Filling a clipping region 3-30
Figure 3-17 Shrinking images between graphics ports 3-33
Figure 3-18 Forty-five degree angles as returned by the `PtToAngle` procedure 3-57
Figure 3-19 Oval width and height in rounded rectangles 3-63
Figure 3-20 Using angles to define the radii for arcs and wedges 3-72
Figure 3-21 Using `PaintArc` to paint a 45° angle 3-74
Figure 3-22 Framing and painting polygons 3-82
Figure 3-23 Using `ScalePt` and `MapPt` 3-105
Figure 3-24 A source image and its resulting mask produced by the `SeedFill` procedure 3-109
Figure 3-25 Parameters for the `SeedFill` and `CalcMask` procedures 3-110
Figure 3-26 A source image and the resulting mask produced by the `CalcMask` procedure 3-111
Figure 3-27 Using `CopyBits` to stretch an image 3-113
Figure 3-28 Standard patterns 3-128
Figure 3-29 Format of a compiled pattern (`'PAT '`) resource 3-140
Figure 3-30 Format of a compiled pattern list (`'PAT#'`) resource 3-141

Table 3-1 Effect of Boolean transfer modes on 1-bit pixels 3-9
Table 3-2 The global variables for five predefined bit patterns 3-20
Table 3-3 QuickDraw routines for calculating and manipulating rectangles 3-31
Table 3-4 QuickDraw routines for calculating and manipulating regions 3-32

Listing 3-1 Drawing lines with the `LineTo` and `Line` procedures 3-18
Listing 3-2 Using the `PenSize` procedure 3-20
Listing 3-3 Using the `PenPat` procedure to change the pattern of the graphics pen 3-21
Listing 3-4 Using the `FrameRect` procedure to draw rectangles 3-23
Listing 3-5 Using the `PaintRect` and `FillRect` procedures 3-24
Listing 3-6 Using the `FrameOval` procedure to draw ovals 3-25
Listing 3-7 Using the `FrameArc` and `PaintArc` procedures 3-26
Listing 3-8 Creating and drawing a region 3-28
Listing 3-9 Creating a clipping region and filling it with a pattern 3-29
Listing 3-10 Creating a triangular polygon 3-30
Listing 3-11 Using the `CopyBits` procedure to copy between two windows 3-33

Chapter 4 Color QuickDraw 4-1

Figure 4-1 The color graphics port 4-7
Figure 4-2 The pixel map 4-10
Figure 4-3 Translating a 48-bit `RGBColor` record to an 8-bit pixel value on an indexed device 4-14
Figure 4-4 Translating an 8-bit pixel value on an indexed device to a 48-bit `RGBColor` record 4-15
Figure 4-5 Translating a 48-bit `RGBColor` record to a 32-bit pixel value on a direct device 4-15
Figure 4-6 Translating a 48-bit `RGBColor` record to a 16-bit pixel value on a direct device 4-16
Figure 4-7 Translating a 32-bit pixel value to a 48-bit `RGBColor` record 4-16
Figure 4-8 Translating a 16-bit pixel value to a 48-bit `RGBColor` record 4-17
Figure 4-9 Drawing with two different foreground colors (on a grayscale screen) 4-23
Figure 4-10 Using ResEdit to create a pixel pattern resource 4-24
Figure 4-11 Painting and filling rectangles with pixel patterns 4-25
Figure 4-12 Copying pixel images with the `CopyBits` procedure 4-27
Figure 4-13 Copying pixel images with the `CopyMask` procedure 4-29
Figure 4-14 Copying pixel images with the `CopyDeepMask` procedure 4-31
Figure 4-15 Difference between highlighting and inverting 4-42
Figure 4-16 Format of a compiled pixel pattern (`'ppat'`) resource 4-103
Figure 4-17 Format of a compiled color table (`'clut'`) resource 4-104
Figure 4-18 Format of a compiled color icon (`'cicn'`) resource 4-106

Table 4-1 Boolean source modes with colored pixels 4-33
Table 4-2 Arithmetic modes in a 1-bit environment 4-41
Table 4-3 Initial values in the `CGrafPort` record 4-64

Table 4-4 The colors defined by the global variable `QDColors` 4-71
Table 4-5 The default color tables for grayscale graphics devices 4-92
Table 4-6 The default color tables for color graphics devices 4-93

Listing 4-1 Using the Window Manager to create a color graphics port 4-20
Listing 4-2 Changing the foreground color 4-22
Listing 4-3 Rez input for a pixel pattern resource 4-24
Listing 4-4 Using pixel patterns to paint and fill 4-25
Listing 4-5 Using `CopyBits` to produce coloration effects 4-35
Listing 4-6 Setting the highlight bit 4-42
Listing 4-7 Using highlighting for text 4-43

Chapter 5 Graphics Devices 5-1

Figure 5-1 The `GDevice` record 5-5

Listing 5-1 Using the `DeviceLoop` procedure 5-8
Listing 5-2 Drawing into different screens 5-9
Listing 5-3 Zooming a window 5-10

Chapter 6 Offscreen Graphics Worlds 6-1

Listing 6-1 Using a single offscreen graphics world and the `CopyBits` procedure 6-5
Listing 6-2 Using two offscreen graphics worlds and the `CopyMask` procedure 6-10

Chapter 7 Pictures 7-1

Figure 7-1 A picture of a party hat 7-4
Figure 7-2 The `Picture` record 7-5
Figure 7-3 A simple picture 7-12
Figure 7-4 Structure of a compiled picture (`'PICT'`) resource 7-68

Table 7-1 Routine selectors for an application-defined color-picking method 7-61

Listing 7-1 Creating and drawing a picture 7-11
Listing 7-2 Opening and drawing a picture from disk 7-13
Listing 7-3 Replacing QuickDraw's standard low-level picture-reading routine 7-15
Listing 7-4 Determining whether a graphics port is color or basic 7-16
Listing 7-5 A custom low-level procedure for spooling a picture from disk 7-16
Listing 7-6 Pasting in a picture from the scrap 7-17
Listing 7-7 Adjusting the destination rectangle for a picture 7-18
Listing 7-8 Drawing a picture stored in a resource file 7-20
Listing 7-9 Saving a picture as a `'PICT'` file 7-21
Listing 7-10 Replacing QuickDraw's standard low-level picture-writing routine 7-22

Listing 7-11 A custom low-level routine for spooling a picture to disk 7-23
Listing 7-12 Looking for color profile comments in a picture 7-25

Chapter 8 Cursor Utilities 8-1

Figure 8-1 Hot spots in cursors 8-4
Figure 8-2 The standard arrow cursor 8-8
Figure 8-3 The I-beam, crosshairs, plus sign, and wristwatch cursors 8-8
Figure 8-4 A window and its arrow and I-beam regions 8-9
Figure 8-5 Changing the cursor from the I-beam cursor to the arrow cursor 8-10
Figure 8-6 The `'CURS'` resources for an animated globe cursor 8-13
Figure 8-7 An `'acur'` resource for an animated cursor 8-14
Figure 8-8 Format of a compiled cursor (`'CURS'`) resource 8-34
Figure 8-9 Format of a compiled color cursor (`'crsr'`) resource 8-35
Figure 8-10 Format of a compiled animated cursor (`'acur'`) resource 8-37

Table 8-1 Cursor appearance 8-17

Listing 8-1 Initializing the Cursor Utilities 8-6
Listing 8-2 Changing the cursor 8-10
Listing 8-3 Animating a cursor with the `RotateCursor` procedure 8-15
Listing 8-4 Animating a cursor with the `SpinCursor` procedure 8-15

Chapter 9 Printing Manager 9-1

Figure 9-1 A standard File menu for an application 9-5
Figure 9-2 The style dialog box for a StyleWriter printer 9-6
Figure 9-3 The style dialog box for a LaserWriter printer 9-7
Figure 9-4 The job dialog box for a StyleWriter printer 9-7
Figure 9-5 The job dialog box for a LaserWriter printer 9-8
Figure 9-6 Page and paper rectangles 9-10
Figure 9-7 A `TPrint` record 9-12
Figure 9-8 The print status dialog box for a LaserWriter printer driver printing in the background 9-13
Figure 9-9 A status dialog box with the LaserWriter printer driver's print status dialog box 9-14
Figure 9-10 How the `PrJobMerge` procedure works 9-26
Figure 9-11 Sample resolutions for a PostScript printer and a QuickDraw printer 9-30
Figure 9-12 A print job dialog box with additional checkboxes 9-35

Table 9-1 Values for the `lParam1` parameter when using the `iPrDevCtl` control constant 9-83

Listing 9-1 Reading a document's `TPrint` record 9-17
Listing 9-2 A sample printing loop 9-20
Listing 9-3 Checking whether the current printer driver supports the `PrGeneral` procedure 9-29

Listing 9-4 Using the `getRslDataOp` and `setRslOp` opcodes with the `PrGeneral` procedure 9-31
Listing 9-5 Using the `getRotnOp` opcode with the `PrGeneral` procedure to determine page orientation 9-33
Listing 9-6 Using the `draftBitsOp` opcode with the `PrGeneral` procedure for enhanced draft-quality printing 9-34
Listing 9-7 Installing an initialization function to alter the print job dialog box 9-37
Listing 9-8 Adding items to a print job dialog box 9-37
Listing 9-9 An idle procedure 9-40

Appendix A Picture Opcodes A-1

Figure A-1 A picture A-23

Table A-1 Data types for picture opcodes A-4
Table A-2 Opcodes for extended version 2 and version 2 pictures A-5
Table A-3 Opcodes for version 1 pictures A-18

Listing A-1 Data for the `BkPixPat`, `PnPixPat`, and `FillPixPat` opcodes A-17
Listing A-2 Data for the `BitsRect` and `PackBitsRect` opcodes A-17
Listing A-3 Data for the `BitsRgn` and `PackBitsRgn` opcodes A-18
Listing A-4 Creating and drawing an extended version 2 picture A-22
Listing A-5 A decompiled extended version 2 picture A-23
Listing A-6 A decompiled version 2 picture A-24
Listing A-7 A decompiled version 1 picture A-25

Appendix B Using Picture Comments for Printing B-1

Figure B-1 The line layout error between a bitmapped font and a PostScript font B-12
Figure B-2 Major and minor glyphs B-13
Figure B-3 Distributing layout error to the major glyphs B-13
Figure B-4 Distributing layout error among major and minor glyphs B-14
Figure B-5 Using the `LineLayoutOff` and `LineLayoutOn` picture comments B-15
Figure B-6 Variations in text alignment B-18
Figure B-7 Types of polygons B-23
Figure B-8 QuickDraw and PostScript polygons B-29
Figure B-9 Changing the pen width using the `SetLineWidth` picture comment B-36

Table B-1 Names, values, and data sizes for picture comments B-5
Table B-2 Low-level QuickDraw routines disabled by the `PostScriptBegin` comment B-9

Listing B-1 Synchronizing QuickDraw and the PostScript driver B-10
Listing B-2 Flushing the buffer for a PostScript printer driver B-11
Listing B-3 Disabling line layout by using the `LineLayoutOff` and `StringBegin` picture comments B-17

Listing B-4 Displaying rotated text using picture comments B-21
Listing B-5 Creating polygons B-26
Listing B-6 Drawing polygons B-27
Listing B-7 Using picture comments to rotate graphics B-31
Listing B-8 Using the `RotateCenter`, `RotateBegin`, and `RotateEnd` picture comments B-32
Listing B-9 Using the `DashedLine` picture comment B-34
Listing B-10 Using the `SetLineWidth` picture comment B-37
Listing B-11 Sending PostScript code directly to the printer B-39

About This Book

This book, *Inside Macintosh: Imaging With QuickDraw*, describes how to create images, display them in black and white or color, and print them using QuickDraw—the imaging engine available on all Macintosh computers. The chapters in this book and the information they contain are summarized here.

- "Introduction to QuickDraw" introduces you to the terms, concepts, and capabilities of QuickDraw.
- "Basic QuickDraw" describes how to create and manage the basic graphics port—the drawing environment in which your application can create graphics and text in either black and white or eight basic colors.
- "QuickDraw Drawing" explains the routines and data structures—common to both basic QuickDraw and Color QuickDraw—that your application can use to draw lines, rectangles, rounded rectangles, ovals, arcs, wedges, polygons, and regions, and to copy images from one graphics port to another. This chapter also describes the routines that you can use to perform calculations and other manipulations of these shapes—including comparing them and finding their unions and intersections.
- "Color QuickDraw" describes the version of QuickDraw that provides a range of color and grayscale capabilities to your application. You should read this chapter if your application needs to use shades of gray or more colors than the eight predefined colors provided by basic QuickDraw.
- "Graphics Devices" describes how Color QuickDraw manages video devices so that your application can draw to a window's graphics port without regard to the capabilities of the screen—even if the window spans more than one screen.
- "Offscreen Graphics Worlds" describes how you can improve your application's appearance and performance by constructing images in offscreen graphics worlds before drawing them onscreen.
- "Pictures" describes how to create and draw QuickDraw pictures, which are sequences of saved drawing commands that your application can share among documents, even among documents created by other applications.
- "Cursor Utilities" describes how to create and use cursors, including color and animated cursors, for indicating the relative mouse location onscreen and for indicating when your application is performing a lengthy task.
- "Printing Manager" describes how your application can print by creating a printing graphics port and drawing into it using QuickDraw routines. This chapter also explains how to implement the user interface that helps users when they wish to print.

- Appendix A, "Picture Opcodes," describes opcodes used by the `DrawPicture` procedure to determine what object to draw or what mode to change for subsequent drawing. Your application generally should not read or write picture opcodes directly but should instead use the QuickDraw routines described in the chapter "Pictures" for generating and processing the opcodes. Picture opcodes are listed in this appendix for your application's debugging purposes.
- Appendix B, "Using Picture Comments for Printing," describes how your application can use picture comments to instruct printer drivers to perform graphics operations that QuickDraw does not support.

Most applications draw in windows, which are rectangular areas that are usually subsets of the screen. Each window represents its own QuickDraw graphics port. When you create a window, the Window Manager uses QuickDraw to create a graphics port in which the window's contents are displayed. This book describes how to draw inside a window. See the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for a complete description of creating and managing the windows inside which your application draws.

In addition to the features described in this book, QuickDraw also provides extensive text support. However, most of that support is not described in this book; for information on handling text in your application, see instead *Inside Macintosh: Text*.

If you are new to programming for the Macintosh computer, you should also read *Inside Macintosh: Overview* for an introduction to general concepts of Macintosh programming.

Format of a Typical Chapter

The chapters in this book follow a standard structure. For example, the chapter "Color QuickDraw" contains these sections:

- "About Color QuickDraw." This section introduces the drawing environment provided by Color QuickDraw, the data structure with which your application specifies colors to Color QuickDraw, and the manner in which Color QuickDraw translates the colors that your application specifies into colors on the user's screen.
- "Using Color QuickDraw." Using code samples and step-by-step instructions, this section describes how to use Color QuickDraw to accomplish the basic tasks necessary for drawing in color.

- "Color QuickDraw Reference." This section provides a complete reference to the data structures, routines, and resources that your application can use to draw with Color QuickDraw. Each routine description also follows a standard format, which presents the routine declaration followed by a description of every parameter of the routine. Some routine descriptions also give additional descriptive information, such as assembly-language information or result codes.
- "Summary of Color QuickDraw." This section provides the Pascal and C interfaces for the constants, data structures, routines, and result codes associated with Color QuickDraw drawing. It also includes relevant assembly-language interface information.

Conventions Used in This Book

Inside Macintosh uses various conventions to present information. Words that require special treatment appear in specific fonts or font styles. Certain information uses special formats so that you can scan it quickly.

Special Fonts

All code listings, reserved words, and names of actual data structures, fields, constants, parameters, and routines are shown in Courier (`this is Courier`).

Words that appear in **boldface** are key terms or concepts and are defined in the glossary.

Types of Notes

There are several types of notes used in this book.

Note
A note like this contains information that is interesting but possibly not essential to an understanding of the main text. (An example appears on page 2-3 in the chapter "Basic QuickDraw.") ◆

IMPORTANT
A note like this contains information that is essential for an understanding of the main text. (An example appears on page 2-4 in the chapter "Basic QuickDraw.") ▲

▲ **WARNING**
Warnings like this indicate potential problems that you should be aware of as you design your application. Failure to heed these warnings could result in system crashes or loss of data. (An example appears on page 2-31 in the chapter "Basic QuickDraw.") ▲

Empty Strings

This book occasionally instructs you to provide an empty string in routine parameters and resources. How you specify an empty string depends on what language and development environment you are using. In Rez input files and in C code, for example, you specify an empty string by using two double quotation marks (""), and in Pascal you specify an empty string by using two single quotation marks ('').

Assembly-Language Information

Inside Macintosh provides information about the trap macro and routine selector for specific routines like this:

Trap macro	**Selector**
`_QDExtensions`	$00E0010

In "Assembly-Language Summary" at the end of a chapter, *Inside Macintosh* presents information about the fields of data structures in this format:

0	`baseAddr`	long	bitmap base address
4	`rowBytes`	word	row bytes (must be even)
6	`bounds`	8 bytes	boundary rectangle

The left column indicates the byte offset of the field from the beginning of the data structure. The second column shows the field name as defined in the MPW assembly-language interface files; the third column indicates the size of that field. The fourth column provides a brief description of the use of the field. For a complete description of each field, see "Data Structures" in the main text of the chapter.

The Development Environment

The system software routines described in this book are available using Pascal, C, or assembly-language interfaces. How you access these routines depends on the development environment you are using. When showing system software routines, this book uses the Pascal interface available with the Macintosh Programmer's Workshop (MPW).

All code listings in this book are shown in Pascal (except for listings that describe resources, which are shown in Rez-input format). They show methods of using various routines and illustrate techniques for accomplishing particular tasks. All code listings have been compiled and, in many cases, tested. However, Apple Computer, Inc., does not intend for you to use these code samples in your application. You can find the location of code listings in the list of figures, tables, and listings. If you know the name of a particular routine (such as `MyDrawLines`) shown in a code listing, you can find the page on which the routine occurs by looking under the entry "sample routines" in the index of this book.

To make the code listings in this book more readable, they show only limited error handling. You need to develop your own techniques for handling errors.

Introduction to QuickDraw

Contents

Drawing Environments 1-4
QuickDraw's Coordinate Plane 1-6
Images 1-10
Colors 1-17
  Indexed Colors 1-19
  Direct Colors 1-20
Multiple Screens 1-21
From Memory Bits to Onscreen Pixels 1-24
From Memory Bits to Printers 1-26
Other Graphics Managers 1-28

This chapter introduces you to the terms, concepts, and capabilities of **QuickDraw,** a collection of system software routines that your application can use to perform most image-manipulation operations on Macintosh computers. This chapter also introduces you to the Printing Manager, which your application can use to print the images you create with QuickDraw.

Imaging entails the construction and display of graphical information. Such graphical information can consist of shapes, pictures, and text, and can be displayed on such output devices as screens and printers. You should read this chapter if you are new to Macintosh programming. The topics introduced in this chapter are explained in detail in the rest of the chapters of this book. However, for information on QuickDraw's text-handling facilities, you should instead see *Inside Macintosh: Text*.

Macintosh system software not only provides enormous imaging flexibility, it also handles much of the programming overhead that such flexibility requires. For example, Color QuickDraw automatically handles multiple screens of differing sizes and capabilities, so that most applications don't need to determine where the user has placed a window or what equipment the user has set up.

This rest of this chapter introduces

- **basic QuickDraw,** the imaging engine for all Macintosh computers
- **Color QuickDraw,** the color imaging system with which your application can display hundreds, thousands, even millions of colors on grayscale and color screens
- the **Printing Manager,** the collection of system software routines that allows your application to communicate with printer drivers to print on any variety of printer
- other imaging managers associated with QuickDraw

QuickDraw is the part of the Macintosh Toolbox that performs graphics operations on the user's screen. All Macintosh applications use QuickDraw indirectly whenever they call other Toolbox managers to create and manage the basic user interface elements (such as windows, controls, and menus, as described in *Inside Macintosh: Macintosh Toolbox Essentials*).

As the Macintosh has evolved toward greater graphics capabilities, QuickDraw has grown along with it. Each new generation of QuickDraw has maintained compatibility with those that preceded it, while adding new capabilities and expanding the range of possible display devices. This evolutionary approach has helped to ensure that existing applications, written for earlier Macintosh models, continue to work as more powerful computers are developed.

The development of QuickDraw has progressed along these three main evolutionary stages:

- Basic QuickDraw, which was designed for the earliest Macintosh models with their built-in black-and-white screens. System 7 added new capabilities to basic QuickDraw, including support for offscreen graphics worlds and the extended version 2 picture format. Basic QuickDraw is still used in more recent black-and-white Macintosh systems such as the Macintosh Classic® and PowerBook 100 computers.
- The original version of Color QuickDraw, which was introduced with the first Macintosh II systems. This first generation of Color QuickDraw could support up to 256 colors.
- The current version of Color QuickDraw, which was originally introduced as 32-Bit Color QuickDraw and is now part of System 7. This version has been expanded to support up to millions of colors.

Applications that use only basic QuickDraw routines are compatible with all Macintosh systems. However, applications that use routines specific to Color QuickDraw cannot run on computers supporting only basic QuickDraw.

Drawing Environments

The Macintosh computer was the first to popularize the bitmapped screen, as opposed to the character-oriented screen—common to terminals and early personal computers—on which only a single, built-in character set could be displayed. On a bitmapped screen every pixel can be manipulated. **Pixels** are the dots that make up a visible image on the screen. Drawing on the Macintosh screen consists of manipulating memory bits that QuickDraw translates into an analogous manipulation of pixels. This allows your application to create shapes and characters in differing sizes and differing styles. Such flexibility gives your application and its users many of the capabilities of a design studio and a print shop.

Your application performs all graphics operations in graphics ports. A **graphics port** is a drawing environment—defined by a `GrafPort` record for a basic graphics port or a `CGrafPort` record for a color graphics port—that contains the information QuickDraw needs to transmit drawing operations from bits in memory to onscreen pixels.

A **basic graphics port** is the drawing environment provided by basic QuickDraw; a basic graphics port contains the information that basic QuickDraw uses to create and manipulate onscreen either black-and-white images or color images that employ a basic eight-color system.

A **color graphics port** is the sophisticated color drawing environment provided by Color QuickDraw; a color graphics port contains the information that Color QuickDraw uses to create and manipulate grayscale and color images onscreen.

While your application can draw directly into basic and color graphics ports, you can improve your application's appearance and performance by constructing images in offscreen graphics worlds and then copying them to onscreen graphics ports. An **offscreen graphics world** is a sophisticated environment for preparing complex color, grayscale, or black-and-white images before displaying them on the screen. Defined in a private data structure referred to by a pointer of type `GWorldPtr`, an offscreen graphics world also contains a graphics port of its own.

Your application can print the images it prepares in graphics ports by drawing into a printing graphics port using QuickDraw drawing routines. A **printing graphics port** is the printing environment defined by a `TPrPort` record, which contains a graphics port plus additional information used by the printer driver and system software.

The visible image for a graphics port is contained in either a bitmap or a pixel map. A **bitmap** is defined by a data structure of type `BitMap`, and it represents the positions and states of a corresponding set of pixels, which can be either black and white or the eight predefined colors provided by basic QuickDraw. A bitmap is contained within a basic graphics port. A **pixel map** is defined by a data structure of type `PixMap`, and it represents the positions and states of a corresponding set of color pixels. A handle to a pixel map is contained within a color graphics port.

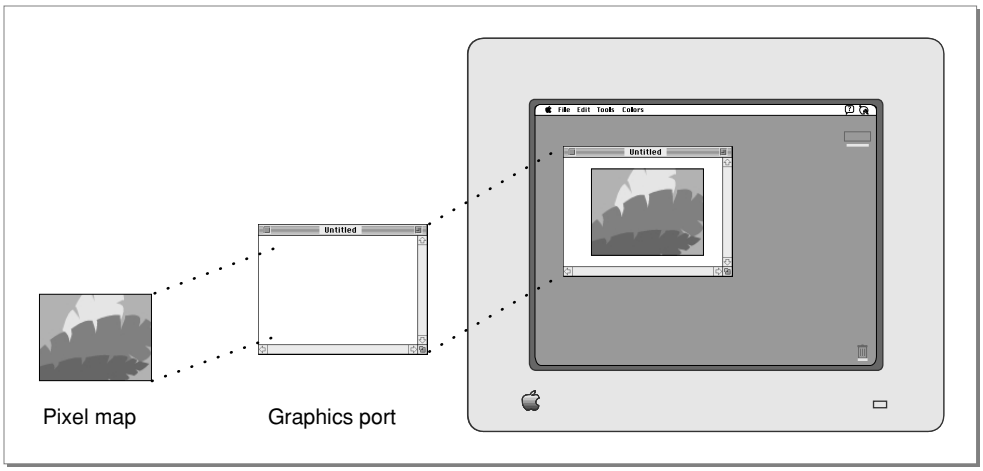
Because your application can typically deal with graphics ports instead of hardware devices, QuickDraw helps your application achieve device independence. A graphics port does the following:

- It specifies the bitmap or pixel map that points to the area of memory in which your drawing operations take place. The bitmap or pixel map contains information about how that memory should be arranged to map bits to the screen.
- It contains a metaphorical graphics pen with which to perform drawing operations. You can set this pen to different sizes, patterns, and colors.
- It holds information about text: if your application or its user writes characters, they will be styled and sized according to information in your application's graphics port.

The fields of a graphics port are maintained by QuickDraw, and you should never write directly into those fields. However, QuickDraw provides routines for changing the fields of a graphics port: you can point to an image in a different area of memory, reshape and resize the pen, change the pen's pattern and color, and switch fonts. You can, and often must, read the fields of a graphics port.

Figure 1-1 illustrates how an onscreen image represents bits in memory. In this figure, an application uses the Window Manager function `GetNewCWindow` to create a new window on a grayscale screen; `GetNewCWindow` automatically uses Color QuickDraw to create the graphics port for the empty window. The application uses Color QuickDraw procedures to fill the graphics port's pixel map with an image and to draw the image onscreen.

Figure 1-1 A grayscale image representing bits in memory

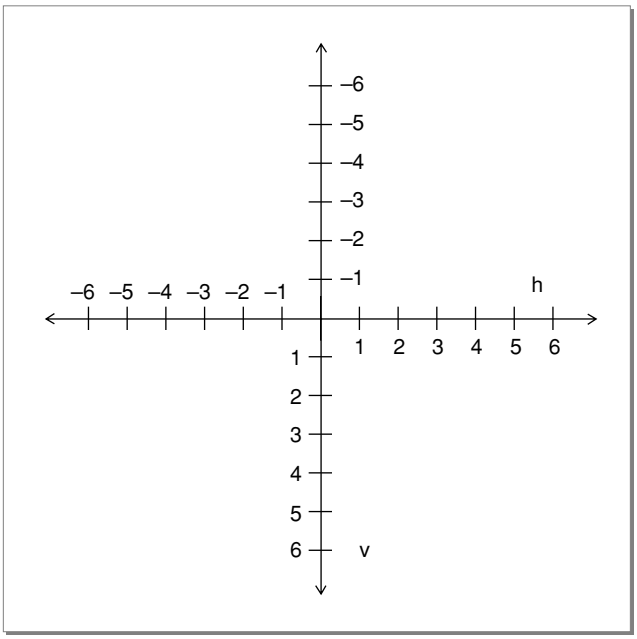


QuickDraw's Coordinate Plane

Your application typically uses Window Manager routines to create graphics ports in the form of windows. Your application can draw into a window without regard to its location on the screen—even if the window spans more than one screen. This is possible because QuickDraw maintains a global coordinate system for a computer's entire potential drawing space and a different local coordinate system for every window displayed in this space.

The Macintosh screen (or screens) on which QuickDraw displays your images represents a small part of a large global coordinate plane. Coordinates in the **global coordinate system** reflect the entire potential drawing space on this plane. The (0,0) origin point of the global coordinate plane is assigned to the upper-left corner of the main screen—that is, the one with the menu bar—while coordinate values increase to the right and (unlike a Cartesian plane) down. Any pixel on the screen can be specified by a vertical coordinate (ordinarily labeled v) and a horizontal coordinate (ordinarily labeled h). Figure 1-2 illustrates the QuickDraw global coordinate plane.

Figure 1-2 The QuickDraw global coordinate plane



Note
The orientation of the vertical axis, while convenient for computer graphics, differs from mathematical convention. Also, the coordinate plane is bounded by the limits of QuickDraw coordinates, which range from –32768 to 32767. ◆

Windows are rectangular areas that are subsets of the global coordinate plane. Each window represents its own QuickDraw graphics port. When you create a window, the Window Manager uses QuickDraw to create a graphics port in which the window's contents are displayed. (See the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for a complete description of creating and managing windows.)

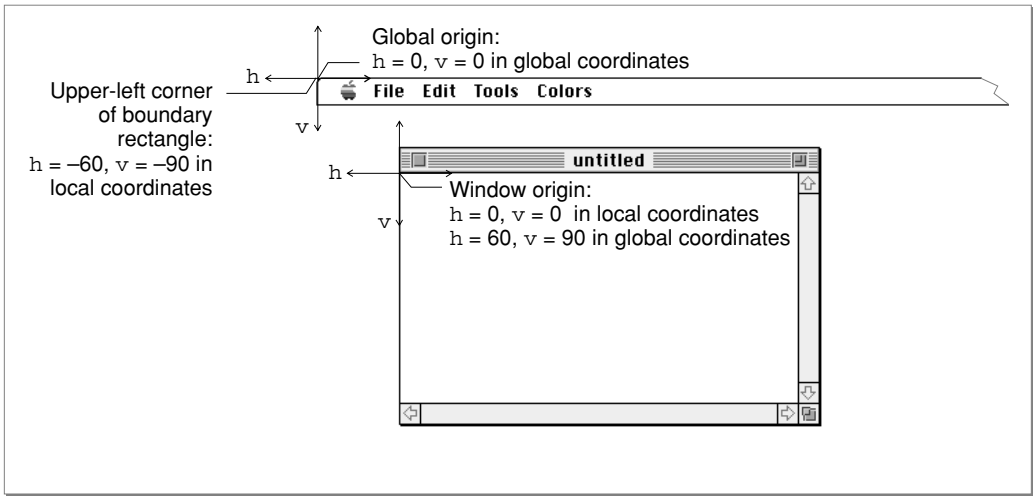
When your application creates a new graphics port, QuickDraw defines a **boundary rectangle,** which, by default, is the entire main screen; this rectangle links the local coordinate system of a graphics port to QuickDraw's global coordinate system and defines the area of the pixel image or bit image into which QuickDraw can draw. A boundary rectangle is stored in the pixel map for a color graphics port or in the bitmap for a basic graphics port.

The graphics port includes a field called `portRect`, which defines a rectangle to be used for drawing. In a graphics port that represents a window, the `portRect` rectangle—or simply, the **port rectangle**—represents the window's content region.

When you use the Window Manager to place a window on the screen, you specify the location of its port rectangle in global coordinates. However, within the port rectangle, the drawing area is described using a **local coordinate system.** You draw into a window

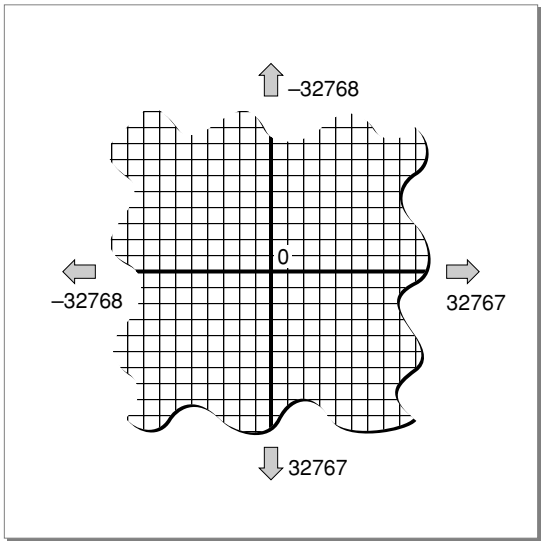
in local coordinates, without regard to the window's location on the screen. Figure 1-3 illustrates the local and global coordinate systems for a sample window that is 180 pixels high by 300 pixels wide, placed with its window origin 90 pixels down and 60 pixels to the right of the upper-left corner of the main screen.

Figure 1-3 A window's local and global coordinate systems



It is helpful to conceptualize the global coordinate plane as a two-dimensional grid—one with integer coordinates ranging from –32768 to 32767—as illustrated in Figure 1-4.

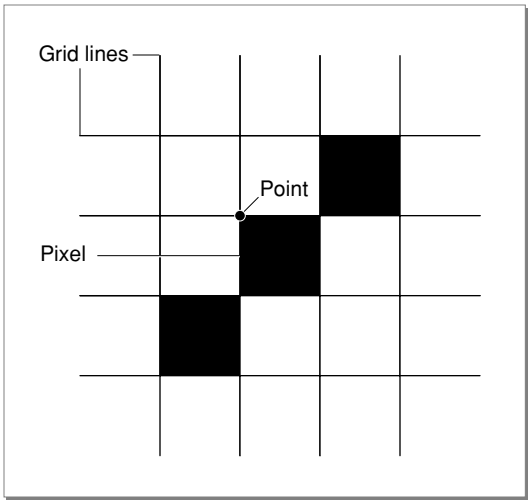
Figure 1-4 The coordinate plane



The intersection of a horizontal and a vertical grid line marks a **point** on the coordinate plane. Notice that, because all coordinates are limited to simple integers, the QuickDraw plane is finite, although very large: there are over four billion points on the grid, many more than there are dots on any screen. When using QuickDraw, you associate small parts of the grid with areas on the screen.

Note also the distinction between points on the coordinate grid and pixels, the dots that make up a visible image on the screen. Figure 1-5 illustrates the relationship between the two: the pixel is down and to the right of the point by which it is addressed.

Figure 1-5 Points and pixels



As the grid lines are infinitely thin, so a point is infinitely small. Pixels, by contrast, lie *between* the lines of the coordinate grid, not at their intersections. This gives them a definite physical extent, so that they can be seen on the screen.

After your application creates a window, the user can move or resize it within the global coordinate system. However, to draw images into the window, your application needs only to specify pixels within the local coordinate system for that window.

Your application uses a cursor to allow the user to select all or part of the content of a window. A **cursor** is a 16-by-16 pixel image that appears on the screen. Called the *pointer* in Macintosh user manuals, the user controls the cursor with the mouse. Basic QuickDraw supplies a predefined cursor for your application: the arrow cursor, which is familiar to all Macintosh users. Your application can also use other cursors. For example, when the user moves the cursor to any text in your application, your application should change the arrow cursor to an I-beam cursor and, when performing a lengthy process that precludes the user from interacting with your application, your application should change the cursor to a wristwatch cursor or an animated cursor.

One point in the cursor's image is designated as the **hot spot,** which in turn points to a location on the screen. The hot spot is the portion of the cursor that must be positioned over a screen object before mouse clicks can have an effect on that object. For example, when the user presses the mouse button, the Event Manager function `WaitNextEvent` reports the location of the cursor's hot spot in global coordinates. Your application can use the basic QuickDraw procedure `GlobalToLocal` to convert the global coordinates of that point to local coordinates for the window.

Images

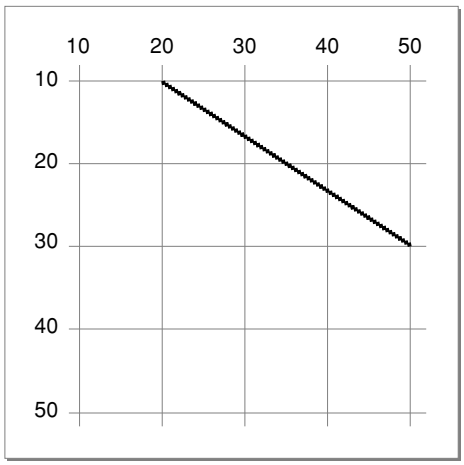
QuickDraw provides a plethora of routines for drawing different kinds of images. These routines typically require that you start at a particular location in a graphics port and then move the graphics pen. The **graphics pen** is a metaphorical device for performing drawing operations onscreen. Your application can set this pen to different sizes, patterns, and colors.

You specify where to begin drawing by placing the pen at some location in the window's local coordinate system, and then specifying an act of drawing, usually from there to another location. Take, for example, the following two lines of code:

```
MoveTo(20,10);
LineTo(50,30);
```

The `MoveTo` procedure places the graphics pen at a point with a horizontal coordinate of 20 and a vertical coordinate of 10 in the local coordinate system of the graphics port, and the `LineTo` procedure draws a line from there to a point with a horizontal coordinate of 50 and a vertical coordinate of 30, as shown in Figure 1-6.

Figure 1-6 Drawing a line



Whenever you draw into a graphics port, the characteristics of its graphics pen determine how the drawing looks. These characteristics are

- pen location, specified in a graphics port's local coordinates
- pen size, specified by a width and height in pixels
- pen pattern, which defines, in effect, the ink that the pen draws with, ranging from solid black to intricate, multicolor patterns
- pattern mode, also called *transfer mode,* which specifies how the pen pattern interacts with white or any existing drawing that the pattern overlays
- pen visibility, specified by an integer indicating whether drawing operations will actually appear—for example, for 0 or negative values, the pen draws with "invisible" ink

The pen's initial settings are a size of 1 pixel by 1 pixel, a solid black pattern, a pattern mode in which the black writes over everything, and visible ink.

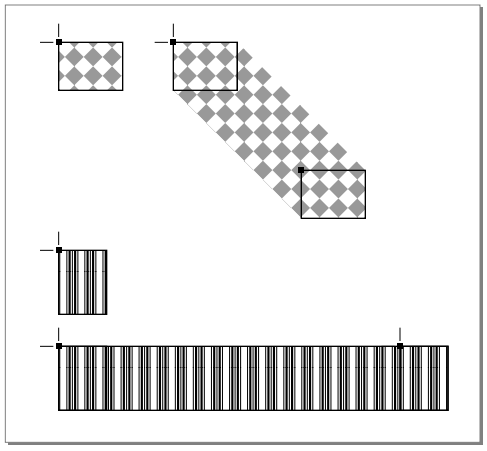
A **pattern** is an image that can be repeated indefinitely to form a repeating design, such as stripes, when drawing lines and shapes or when filling areas on the screen. There are two type of patterns: bit patterns and pixel patterns. A **bit pattern** is an 8-by-8 pixel image drawn by default in black and white, although any two colors can be used on a color screen. A **pixel pattern** can use additional colors and can be of any width and height that's a power of 2.

By starting at a particular position and moving the graphics pen, you can use QuickDraw procedures to define and directly draw a number of graphic shapes using the size and pattern of the graphics pen. Several procedures and the shapes they produce are listed here.

Procedure	Resulting shape
`LineTo`	Line
`DrawChar`	Character
`FrameRect`	Rectangle
`FrameOval`	Oval
`FrameRoundRect`	Rounded rectangle
`FrameArc`	Arc

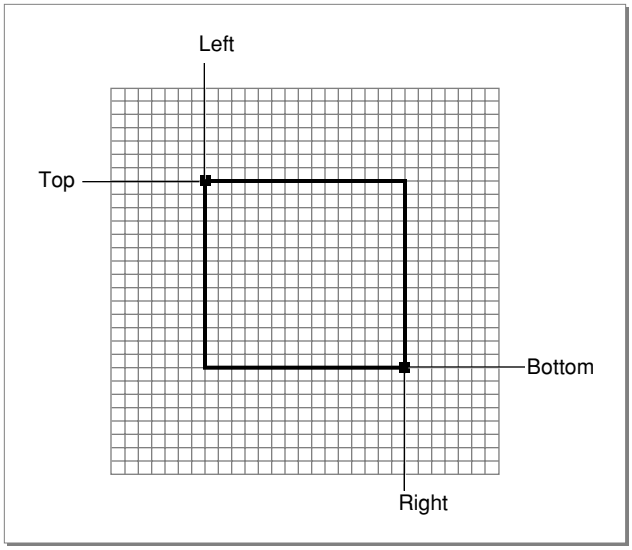
A **line** is defined by two points: the current location of the graphics pen and its destination. The pen hangs below and to the right of the defining points, as shown in Figure 1-7, where two lines are drawn with different bit patterns and pen sizes.

Figure 1-7 Lines drawn with different bit patterns and pen sizes



A **rectangle** can be defined by two points—its upper-left and its lower-right corners, as shown in Figure 1-8, or by four boundaries—its upper, left, lower, and right sides. Rectangles are used to define active areas on the screen, to assign coordinate systems to graphical entities, and to specify the locations and sizes for various graphics operations.

Figure 1-8 A rectangle



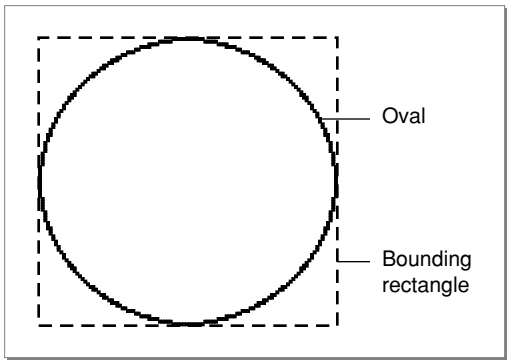
QuickDraw also provides routines that allow you to perform a variety of mathematical calculations on rectangles—changing their sizes, shifting them around, and so on.

Note
Purely speaking, rectangles and points are mathematical entities that have no direct representation on the screen. The borders of the rectangle are infinitely thin, lying along the lines of the coordinate grid. To give a rectangle a shape that can be drawn on the screen, you must use one of QuickDraw's drawing routines, such as `FrameRect` and `PaintRect`. ◆

You use rectangles known as **bounding rectangles** to define the outmost limits of other shapes, such as ovals and rounded rectangles. The lines of bounding rectangles completely enclose the shapes they bound; in other words, no pixels from these shapes lie outside the infinitely thin lines of the bounding rectangles.

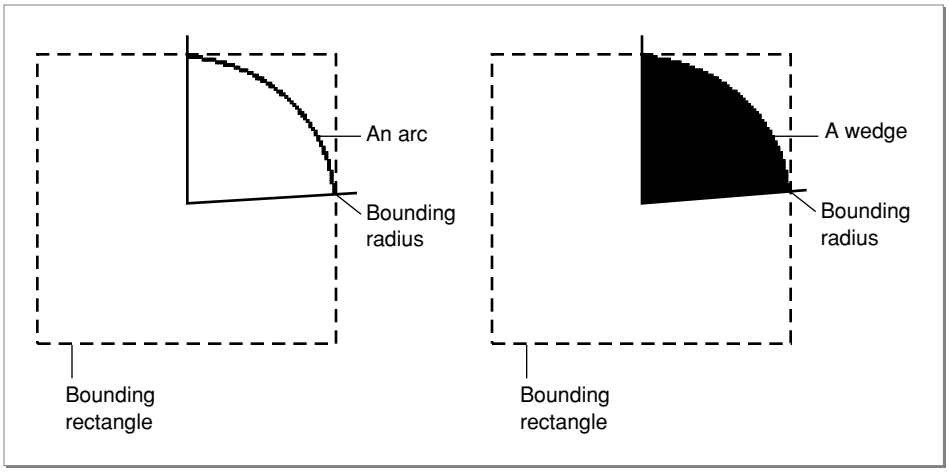
An **oval** is a circular or elliptical shape defined by the bounding rectangle that encloses it. If the bounding rectangle is square (that is, has equal width and height), then the oval is a circle, as shown in Figure 1-9.

Figure 1-9 An oval



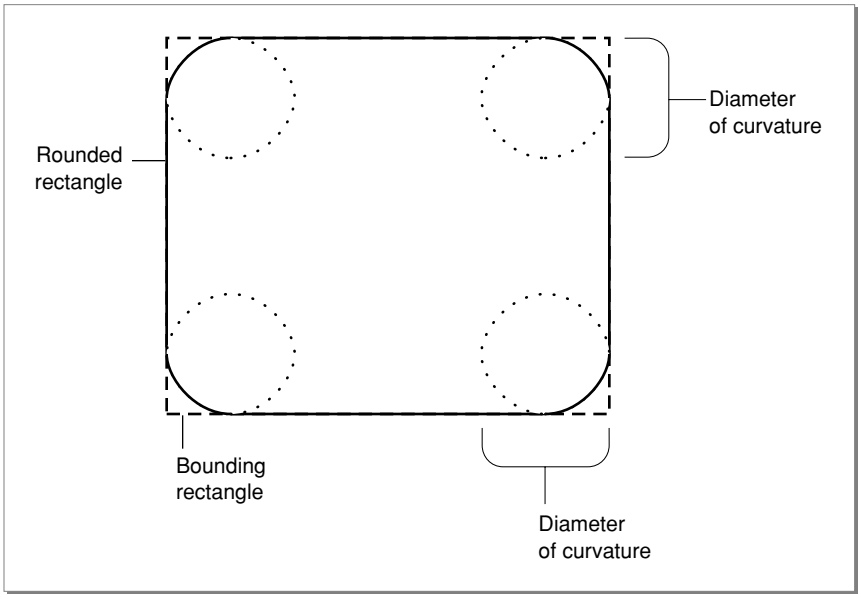
An **arc** is a portion of the circumference of an oval bounded by a pair of radii joining at the oval's center; an arc does not include the bounding radii or any part of the oval's interior. A **wedge** is a pie-shaped segment of an oval, bounded by a pair of radii joining at the oval's center; a wedge does include part of the oval's interior. Arcs and wedges are defined by the bounding rectangle that encloses the oval, along with a pair of angles marking the positions of the bounding radii, as shown in Figure 1-10.

Figure 1-10 An arc and a wedge



A **rounded rectangle** is a rectangle with rounded corners. The figure is defined by the rectangle itself, along with the width and height of the ovals forming the corners (called the *diameters of curvature*), as shown in Figure 1-11. The corner width and corner height are limited to the width and height of the rectangle itself; if they are larger, the rounded rectangle becomes an oval.

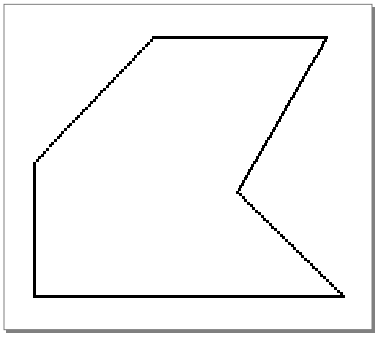
Figure 1-11 A rounded rectangle



Three types of graphic objects—polygons, regions, and pictures—require you to call several routines to create and draw them. You begin by calling a routine that collects drawing commands into a definition for the object. You use a series of drawing routines to define the object. Then you use a routine that signals the end of the object definition. Finally, you use a routine that draws your newly defined object.

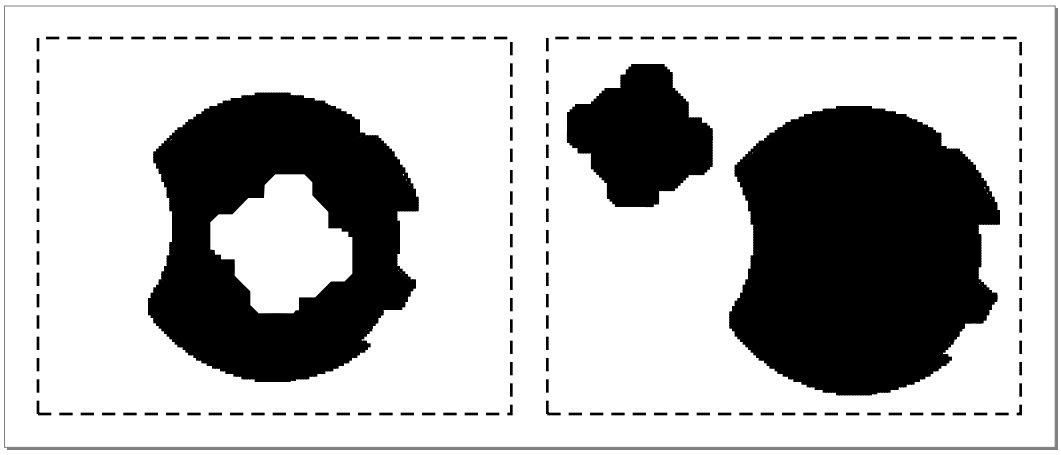
A **polygon** is defined by any sequence of points representing the polygon's vertices, connected by straight lines from one point to the next. You define a polygon by drawing the lines with QuickDraw line-drawing operations: you move to the first vertex point in the sequence, draw a line from there to the second vertex, from there to the third, and so on. When you finish, QuickDraw automatically completes the figure with a closing line from the last vertex back to the first. Figure 1-12 shows an example of a polygon.

Figure 1-12 A polygon



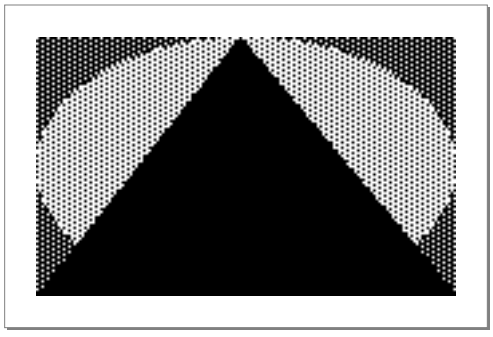
A **region** is an arbitrary area or set of areas, the outline of which is one or more closed loops. One of QuickDraw's most powerful capabilities is the ability to work with regions of arbitrary size, shape, and complexity. You define a region by drawing its boundary with QuickDraw operations. The boundary can be any set of lines and shapes (even including other regions) forming one or more closed loops. A region can be concave or convex, can consist of one connected area or many separate ones, and can even have holes in the middle. In Figure 1-13 the region on the left has a hole and the one on the right consists of two unconnected areas.

Figure 1-13 Two regions



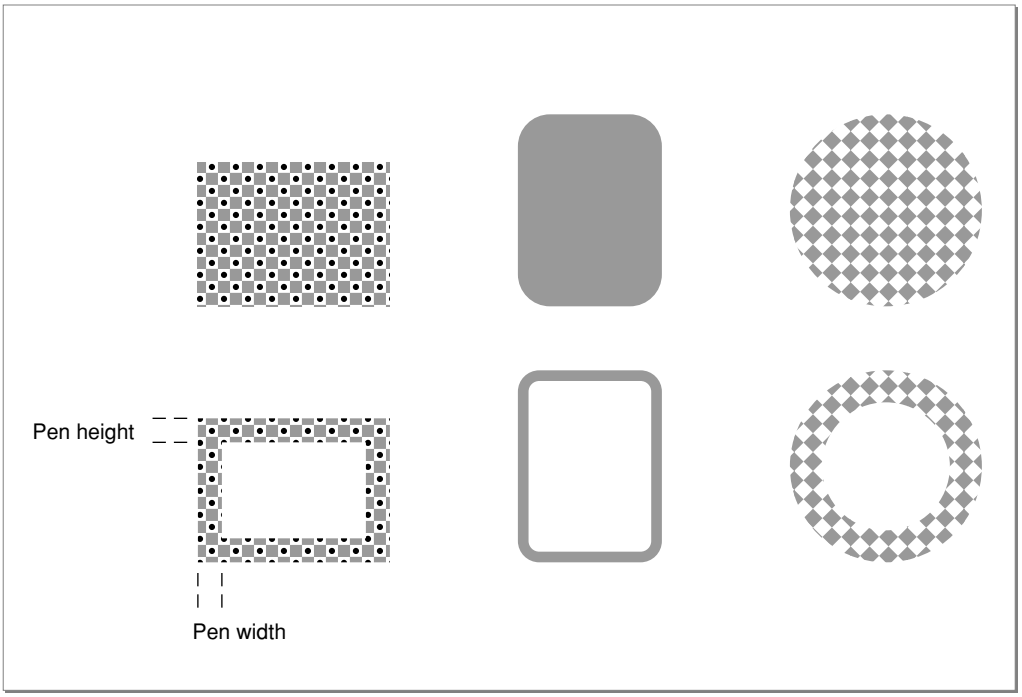
Your application can record a sequence of QuickDraw drawing operations in a **picture** and play its image back later. Pictures provide a form of graphic data exchange: one program can draw something that was defined in another program, with great flexibility and without having to know any details about what's being drawn. Figure 1-14 shows an example of a picture containing a rectangle, an oval, and a rectangle.

Figure 1-14 A simple QuickDraw picture



As you see in Figure 1-14, QuickDraw shapes may be drawn using various pen patterns. Painting a shape fills both its outline and its interior with the current pen pattern. Filling a shape fills both its outline and its interior with any specified pattern (not necessarily the current pen pattern). The three figures in the top row of Figure 1-15 are filled.

Figure 1-15 Filling and framing various shapes



Framing a shape draws just its outline, using the current pen size, pen pattern, and pattern mode. The interior of the shape is unaffected, allowing previously existing pixels to "show through." The three figures in the bottom row of Figure 1-15 are framed. Erasing a shape fills both its outline and its interior with the current background pattern (typically solid white on a black-and-white monitor or a solid background color on a color monitor). Inverting a shape reverses the colors of all pixels within its boundary—for example, all white pixels become black, and all black pixels become white. With color pixels, the results of inverting are less predictable.

Colors

The earliest Macintosh models all used basic QuickDraw to draw to built-in screens with known characteristics. The Macintosh II computer introduced Color QuickDraw, which supports a variety of screens of differing sizes and color capabilities. With Color QuickDraw, users can choose from a wide range of screen options, from simple 12-inch

black-and-white screens to full-page grayscale monitors to large two-page displays capable of presenting millions of colors. Users can even connect two or more separate screens to the same computer and simultaneously view different portions of the system's global coordinate plane.

A pixel, which is short for *picture element*, is the smallest dot that QuickDraw can draw. On a black-and-white monitor, a pixel is a single-color phosphor dot that displays in two states—black and white. On a color screen, three phosphor dots (red, green, and blue) compose each color pixel.

A pair of fields in a graphics port, `fgColor` and `bkColor`, specify a foreground and background color. The **foreground color** is the color used for bit patterns and for the graphics pen when drawing. By default, the foreground color is black. The **background color** is the color of the pixels in the bitmap or pixel map wherever no drawing has taken place. By default, the background color is white. However, when there is a color screen your application can draw with a color other than black by changing the foreground color, and your application can draw into a background other than white by changing the background color. For example, by changing the foreground color to red and the background color to blue before drawing a rectangle, your application can draw a red rectangle against a blue background.

On a color screen, you can draw in color even when you are using a basic graphics port. Although basic QuickDraw graphics routines were designed for black-and-white drawing, they also support an eight-color system that basic QuickDraw predefines for display on color screens and color printers. Because Color QuickDraw also supports this eight-color system, it is compatible across all Macintosh platforms.

The basic QuickDraw color values consist of 1 bit for normal black-and-white drawing (black on white), 1 bit for inverted black-and-white drawing (white on black), 3 bits for the additive primary colors (red, green, blue) used in video display, and 4 bits for the subtractive primary colors (cyan, magenta, yellow, black) used in printing. Basic QuickDraw defines a set of constants for those standard colors:

```
CONST
  blackColor   = 33;
  whiteColor   = 30;
  redColor     = 205;
  greenColor   = 341;
  blueColor    = 409;
  cyanColor    = 273;
  magentaColor = 137;
  yellowColor  = 69;
```

These are the only colors available in basic QuickDraw (or with Color QuickDraw drawing into a basic graphics port).

In Color QuickDraw, however, a color pixel represents up to 48 bits in memory. On a grayscale screen, a white phosphor dot whose intensity can vary is a pixel that usually represents 1, 2, 4, or 8 bits in memory.

To remove (for most applications) the burden of worrying about screen capabilities, Color QuickDraw is device-independent. Your application can use an `RGBColor` **record,** a data structure of type `RGBColor`, to specify a color by its red, green, and blue components, with each component defined as a 16-bit integer. Color QuickDraw compares the resulting 48-bit value with the colors actually available on a video device—such as a plug-in video card or a built-in video interface—at execution time and then chooses the closest match.

What the user finally sees depends on the characteristics of the actual video device and screen. Screens may display color or black and white; the video devices that control them may have indexed colors that support pixels of 1-bit, 2-bit, 4-bit, or 8-bit depths, or direct colors that support pixels of 16-bit or 32-bit depths. Color QuickDraw automatically determines which method is used by the video device and matches your requested color with the closest available color.

Indexed Colors

Some video devices use **indexed colors** to support a maximum of 256 colors at any one time. The indexed color system was created when the Macintosh II computer was introduced, at a time when memory was scarce and moving megabyte images around was impractical. With indexed color, the maximum value of a pixel in a `PixMap` record is limited to a single byte. Each pixel's byte can specify one of 256 (2^8) different values. Video devices implementing indexed color contain a data structure called a **color lookup table** (or, more commonly, a CLUT). The CLUT, in turn, contains entries for all possible color values.

For example, an indexed video device supporting 2 bits per pixel provides indexes into a table of four colors; if two colors are black and white, however, only two other hues can be shown.

An indexed video device supporting 4 bits per pixel can provide indexes into a table of 16 colors, a number sufficient for many straightforward graphics, such as those used in charts, presentations, or games.

An indexed video device supporting a byte (8 bits) for each pixel allows 256 colors to be displayed, which for many images is enough to produce near-photographic quality. The problem is that the colors needed for one near-photographic image may not be appropriate for another. The prevailing shades of browns necessary for displaying a painting by Rembrandt aren't appropriate for the prevailing shades of blues in a Monet painting. Because most indexed video devices use a variable CLUT (rather than a fixed one), you can display a Rembrandt painting with one set of 256 colors, then use system

software to reload the CLUT with a different set of 256 colors for a Monet painting. If your application needs this sort of control on indexed video devices, you can use the Palette Manager (as described in the chapter "Palette Manager" in *Inside Macintosh: Advanced Color Imaging*) to arrange palettes—that is, sets of colors—for particular images and for video devices of differing color capabilities.

Note
Some Macintosh computers, such as grayscale PowerBook computers, have a fixed CLUT, which your application cannot change. ◆

If your application uses a 48-bit `RGBColor` record to specify a color, the Color Manager examines the colors available in the CLUT on the video device. If the video device supports 8 bits per pixel, for example, it contains a CLUT with 256 entries. Comparing these entries to the `RGBColor` record you specify, the Color Manager determines which color in the CLUT is closest, and the Color Manager gives Color QuickDraw the index to this color. Color QuickDraw then draws with this color.

Direct Colors

Video devices that implement **direct color** eliminate the competition for limited table spaces and remove the need for color table matching. By using direct color, video devices may support a maximum of thousands or millions of colors. When you specify a 48-bit `RGBColor` record, Color QuickDraw truncates the least significant bits of its red, green, and blue components to either 16 bits (5 bits each for red, green, and blue, with 1 bit unused) or 32 bits (8 bits each for red, green, and blue, with 8 bits unused). Using 16 bits, direct video devices can display over 32,000 colors; using 32 bits, direct video devices can display as many as 16 million different colors.

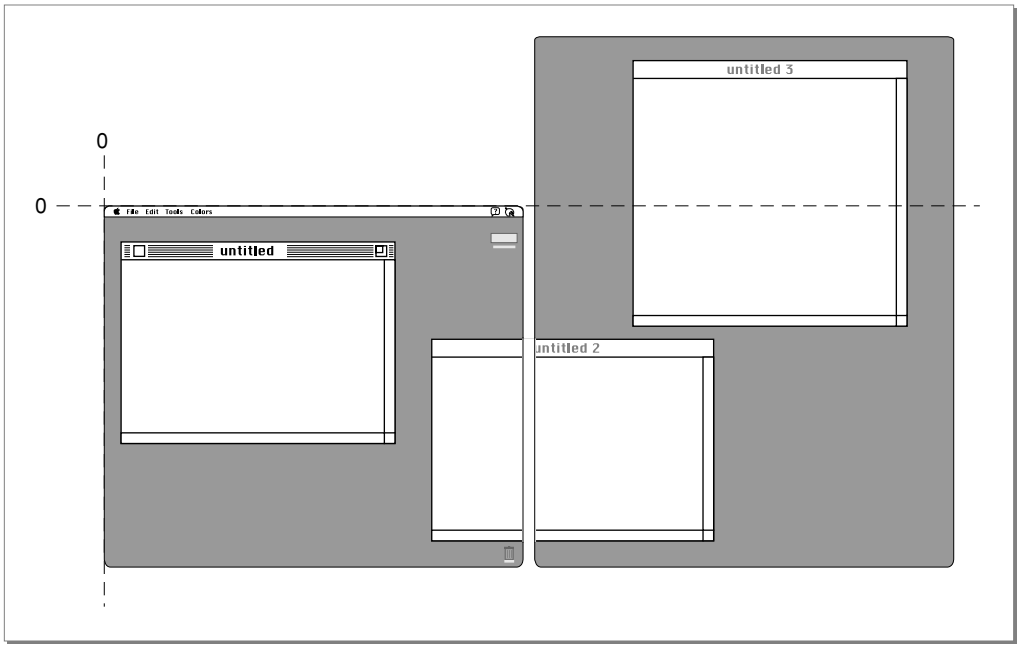
Using direct color not only removes much of the complexity of the CLUT mechanism for video device developers, but it also allows the display of thousands or millions of colors simultaneously, resulting in near-photographic realism.

Multiple Screens

A **video device** is a piece of hardware, such as a plug-in video card or a built-in video interface, that controls a screen. To use more than one screen, a user may have more than one video device installed on his or her computer.

In a drawing environment with multiple screens, the one with the menu bar is the **main screen.** Color QuickDraw maps the (0,0) origin point of the global coordinate plane to the main screen's upper-left corner, and other screens are positioned adjacent to it. In Figure 1-16, a full-page screen sits next to the main screen. Remember that each window—even a window that overlaps two screens—has its own local coordinate system with a (0,0) point at its upper-left corner.

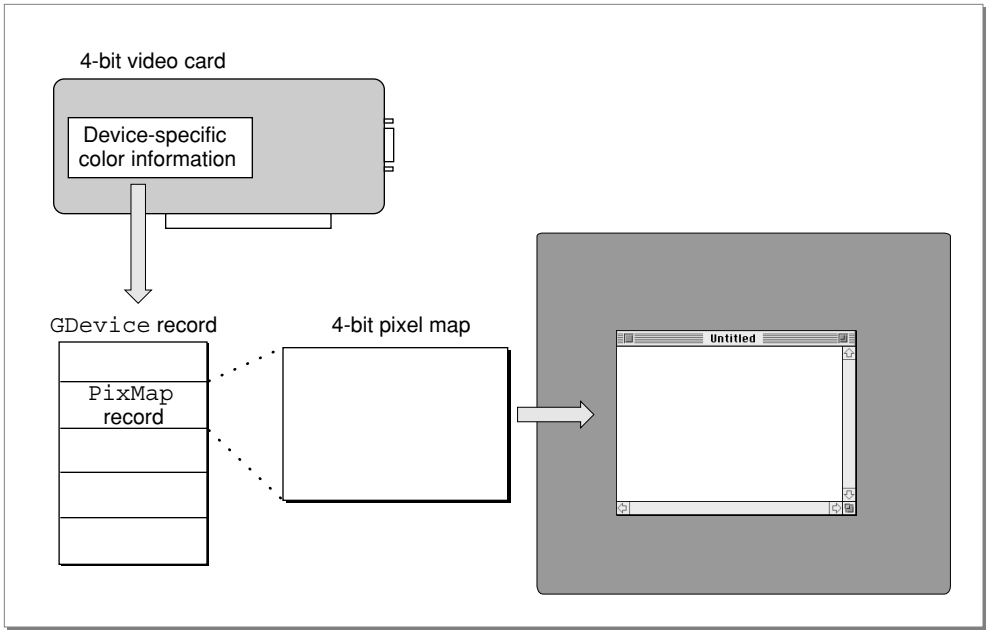
Figure 1-16 A two-screen system



Color QuickDraw stores state information for a video device in a **`GDevice` record.** (Color QuickDraw creates `GDevice` records—basic QuickDraw does not, nor does basic QuickDraw support multiple screens.) When a computer supporting Color QuickDraw starts up, it allocates and initializes a handle to a `GDevice` record for each video device it finds. The firmware in the ROM for each video device supplies information about whether the device uses indexed or direct colors, how much video RAM is available, and so on. Some of this information is stored in the `GDevice` record, where it is available to the entire graphics system.

As illustrated in Figure 1-17, when your application opens a color window, the `CGrafPort` record for the window contains a handle to a `PixMap` record contained in the main screen's `GDevice` record. The `PixMap` record for your window thereby contains the correct pixel specifications for the main screen. Color QuickDraw internally calculates the changes required for drawing to any other screens.

Figure 1-17 The `GDevice` record and pixel map for a 4-bit video card



When a multiscreen system starts up, one of the screens is the **startup screen,** the screen on which the "happy Macintosh" appears. By default, the main screen is the startup screen. However, by using the Monitors control panel, the user can specify a different startup screen.

During the startup of a multiscreen environment, system software calls the Window Manager procedure `InitWindows` to create a region that is the union of all the active screens (minus the menu bar and the rounded corners on the outermost screens). The Window Manager saves this region, called the **gray region,** as the global variable `GrayRgn`. The gray region describes and defines the desktop: the area in which the user can drag windows.

Users can drag windows from one screen to another and even across multiple screens. Color QuickDraw calculates the global coordinates of the rectangle into which it must draw and issues the drawing command to each video device that the rectangle intersects.

For many applications, Color QuickDraw provides a device-independent interface; your application can draw images in a color graphics port for a window, and Color QuickDraw automatically manages the screen display—even if the user has multiple screens. Your application generally never needs to create `GDevice` records. However, you may find it useful for your application to examine `GDevice` records to determine the capabilities of the user's screens. When zooming a window, for example, your application can use `GDevice` records to determine which screen contains the largest area of a window, and then determine the ideal window size for that screen.

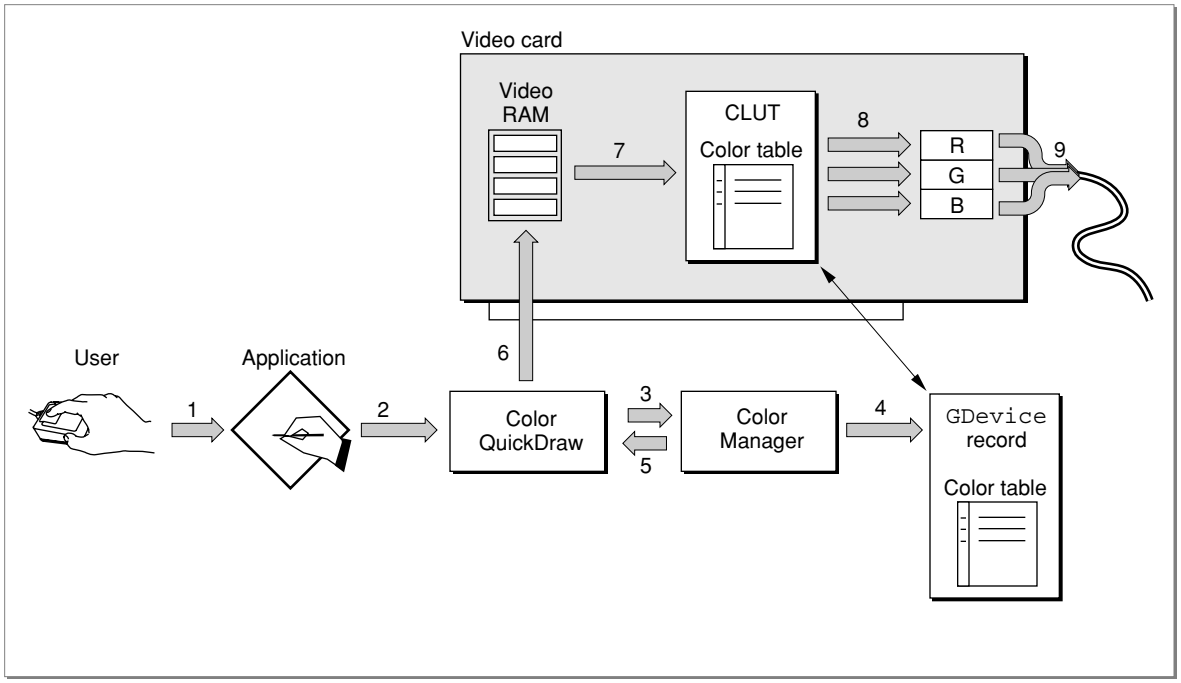
You may also wish to use the `DeviceLoop` procedure to optimize your application's drawing for screens with different capabilities. The `DeviceLoop` procedure searches for video devices that intersect your graphics port's drawing region, and it informs your application of each video device it finds. The `DeviceLoop` procedure provides your application with information about the pixel depth and other attributes of the video device on which drawing is currently taking place. Your application can then choose what drawing technique to use for the current device. When highlighting, for example, your application might invert black and white when drawing onto a 1-bit video device but use magenta as the highlight color on a color screen.

From Memory Bits to Onscreen Pixels

Tracing the path from data in memory to a pixel on one of several connected screens traverses the major elements of QuickDraw and recapitulates much of the discussion in this chapter.

In Figure 1-18, the user of an indexed color system selects a color for some object from an application (1). Using a 48-bit `RGBColor` record to specify the color, the application calls a Color QuickDraw routine to draw the object in that color (2). Color QuickDraw uses the Color Manager to determine what color in the video device's CLUT comes closest to the requested color (3).

Figure 1-18 The indexed-pixel path



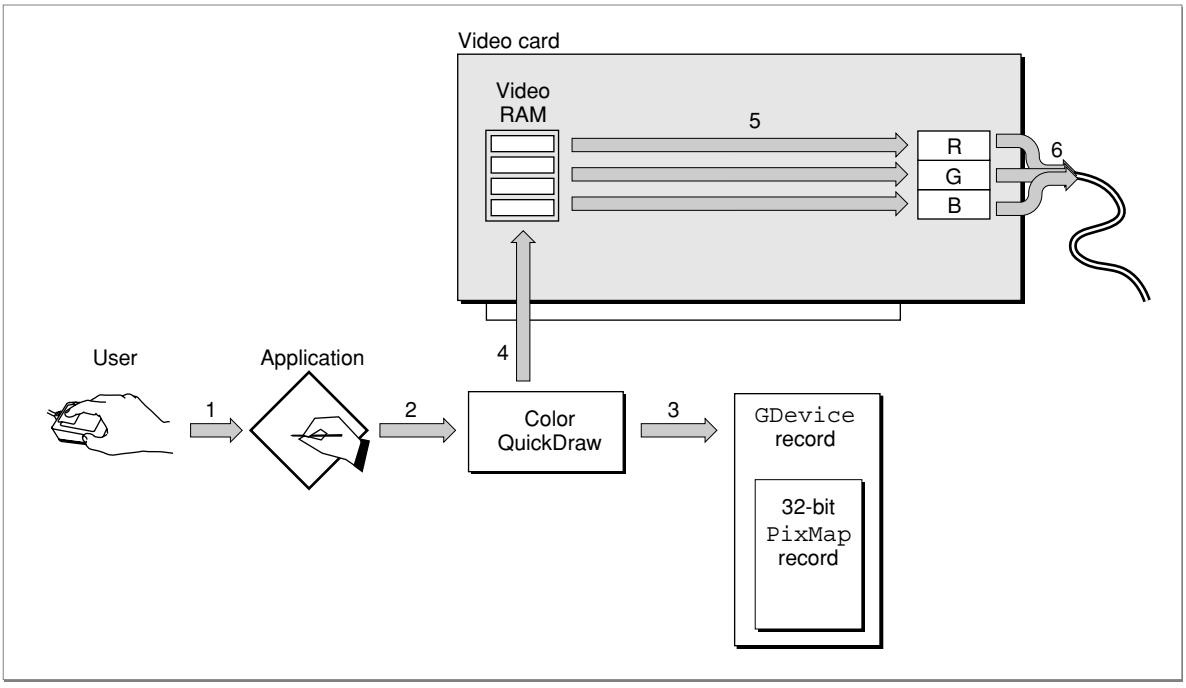
At startup, the video device's declaration ROM supplies information for the creation of the `GDevice` record that describes the characteristics of the device. The resulting `GDevice` record contains a `ColorTable` record that is kept synchronized with the card's CLUT. The Color Manager examines that `GDevice` record to find what colors are currently available (4) and to decide which color comes closest to the one requested by the application. The Color Manager gets the index value for the best match and returns that value to Color QuickDraw (5), which puts the index value into those places in video RAM that store the object (6).

The video device continuously displays video RAM by taking the index values, converting them to colors according to CLUT entries at those indexes (7), and sending them to digital-to-analog converters (8) that produce a signal for the screen (9).

For video devices that support direct color, the Color Manager's selection algorithm isn't needed. When an application specifies a color in an `RGBColor` record, Color QuickDraw uses the most significant 5 or 8 bits of each of the 16-bit red, green, and blue components of the specified color.

Figure 1-19 illustrates a user choosing a color for some object (1). Using a 48-bit `RGBColor` record to specify the color, the application uses a Color QuickDraw routine to draw the object in that color (2). Color QuickDraw knows from the `GDevice` record (3) that the screen is controlled by a direct device in which pixels are 32 bits deep, which means that 8 bits are used for each of the red, green, and blue components of the requested color. Color QuickDraw passes the high 8 bits from each 16-bit component of the 48-bit `RGBColor` record to the video device (4), which stores the resulting 24-bit value in video RAM for the object. The video device continuously displays video RAM by sending the three 8-bit red, green, and blue values for the color to digital-to-analog converters (5) that produce a signal for the screen (6).

Figure 1-19 The direct-pixel path

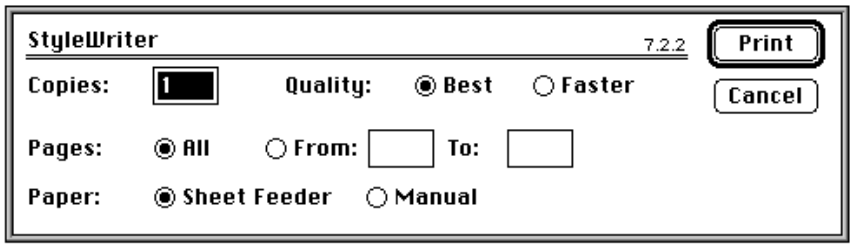


From Memory Bits to Printers

Available on all Macintosh computers, the Printing Manager is a collection of system software routines that your application can use to print to any type of connected printer by using the same QuickDraw routines for printing that your application uses for screen display. You can use the Printing Manager to print documents, to display and alter printing-related dialog boxes, and to handle printing errors. The Printing Manager takes much of the work out of coming up with a single way to handle all possible printer environments.

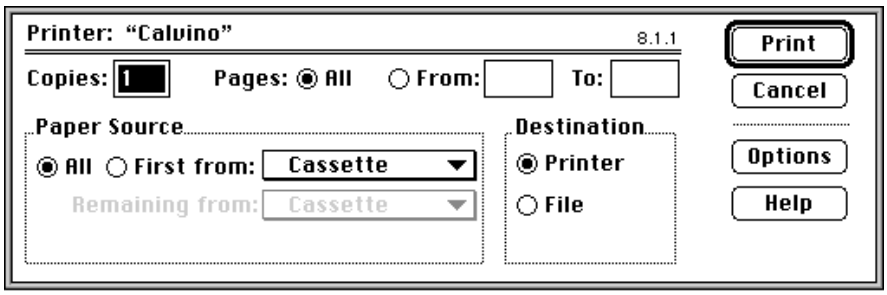
Your application uses the Printing Manager procedure `PrOpen` to open the current printer. (The **current printer** is the printer that the user last selected from the Chooser.) Before printing, your application should display the **job dialog box,** which solicits printing information from the user, such as the number of copies to print, the print quality, and the range of pages to print. Your application can use the `PrJobDialog` procedure to display a job dialog box. The `PrJobDialog` procedure handles all user interaction in the standard dialog items until the user clicks the Print or Cancel button. Figure 1-20 shows an example of a job dialog box. Your application prints the document in the active window if the user clicks the Print button.

Figure 1-20 The job dialog box for a StyleWriter printer



Each printer has its own job dialog box. Thus, a style dialog box for one printer may differ slightly from that of another printer. Figure 1-21 shows a sample job dialog box for a LaserWriter printer.

Figure 1-21 The job dialog box for a LaserWriter printer



A `TPrint` record stores information about the choices made by the user in the print job dialog box. Your application can also customize the job dialog box to ask for additional information.

When the user clicks the Print button in the job dialog box, your application then uses the Printing Manager function `PrOpenDoc` to open a printing graphics port, which consists of a graphics port (either a `GrafPort` or `CGrafPort` record) plus additional information.

To set up the printing graphics port to print a page, your application should call the `PrOpenPage` procedure. Your application then prints by using the QuickDraw routines described in this book to draw into the printing graphics port. The Printing Manager uses a printer driver to do the actual printing. A **printer driver** does any necessary translating of QuickDraw drawing routines and sends the translated instructions and data to the printer. Each type of printer has its own printer driver, which is stored in a resource file in the Extensions folder inside the System Folder. Because your application does not communicate with any of the multitude of available printer drivers but instead uses the Printing Manager to handle this communication, the Printing Manager gives your application device-independent control over the printing process.

When your application has finished drawing into the page set up for the printing graphics port, your application closes the page by using the `PrClosePage` procedure. For every page that the user selects to be printed in a document, your application uses `PrOpenPage` and `PrClosePage`. When your application has finished printing, your application closes the printing graphics port by using the `PrCloseDoc` procedure; your application should then close the Printing Manager by using the `PrClose` procedure.

There are two main types of printer drivers for Macintosh computers: QuickDraw printer drivers and PostScript™ printer drivers. Using QuickDraw drawing operations, **QuickDraw printer drivers** render images on the Macintosh computer and then send the rendered images to the printer in the form of bitmaps or pixel maps. **PostScript printer drivers,** on the other hand, convert QuickDraw drawing operations into equivalent PostScript drawing operations, as necessary. PostScript printers have their own rendering capabilities. PostScript printer drivers typically send drawing operations to the printer, which itself renders images on the page.

For most applications, sending QuickDraw's picture-drawing routines to the printer driver is sufficient: the driver either uses QuickDraw or converts the drawing routines to PostScript. For some applications, such as page-layout programs, this may not be sufficient; such applications may rely on printer drivers to provide several features that are not available, or are difficult to achieve, using QuickDraw.

Using picture comments, your application can instruct printer drivers to perform operations that QuickDraw does not support. Created with the QuickDraw procedure `PicComment`, **picture comments** are data or commands for special processing that can be included in the code an application sends to a printer driver.

A number of picture comments have been given special definitions in printer drivers. The drivers for PostScript printers and even some QuickDraw printers support features unavailable with QuickDraw. When a printer driver encounters one of these comments, it converts it to its own printing code. Other picture comments signal the driver that PostScript code is enclosed, so your application can even include PostScript code directly in the definition of a picture.

Other Graphics Managers

In addition to the QuickDraw routines described in this book, several other collections of system software routines are available to assist you in drawing and printing images.

Your application can use QuickDraw's text-handling routines to measure and draw text ranging in complexity from a single glyph to a line of justified text containing multiple languages and styles. In addition to measuring and drawing text, QuickDraw's text-handling routines also help you to determine which characters to highlight and where to mark the insertion point. These routines are described in the chapter "QuickDraw Text" in *Inside Macintosh: Text*.

To provide more sophisticated color support on indexed graphics devices, your application can use the Palette Manager. The **Palette Manager** allows your application to specify sets of colors that it needs on a window-by-window basis. An indexed device supporting a byte for each pixel allows 256 colors to be displayed. On a video device that uses a variable CLUT, your application can use the Palette Manager to display tens of thousands of palettes—that is, sets of colors—consisting of 256 colors each, so that your application has up to 16 million colors at its disposal (although only 256 different colors

can appear at once). For example, your application can use the Palette Manager to load the CLUT with a set of prevailingly brown colors to display a Rembrandt painting, then reload the CLUT with a set of prevailingly blue colors for a Monet painting. For information about the Palette Manager, see *Inside Macintosh: Advanced Color Imaging*.

To solicit color choices from users, your application can use the Color Picker Utilities. The **Color Picker Utilities** provide your application with a standard dialog box for soliciting a color choice from users. The Color Picker Utilities also provide routines that allow your application to convert between colors specified in `RGBColor` records and colors specified for other color models, such as the CMYK model used by many color printers. Most applications use the Color Picker Utilities only for soliciting color choices. To learn how to use the Color Picker Utilities, see *Inside Macintosh: Advanced Color Imaging*.

As color devices for input and output proliferate, so do the problems of moving images between them with good results. Different device types use different color models, which produce different gamuts, or ranges of colors. Screens, for example, typically display colors as combinations of red, green, and blue—combinations that your application specifies with `RGBColor` records when using Color QuickDraw. Screens by different manufacturers may be capable of displaying different intensities of red, green and blue, so that even though the screens work with RGB colors, their gamuts may be different. Color printers typically use a CMYK color model to work with varying intensities of cyan, magenta, yellow, and black. Print technologies vary drastically, and the gamut that an ink jet color printer can display may be quite different from one based on another technology. A single printer may be able to produce different gamuts depending on the paper or ink in use at the time of printing.

Two devices with differing color gamuts cannot reproduce each other's colors exactly, but shifting the colors of one device may improve the visual match. To match colors between screens and input and output devices such as scanners and printers, Macintosh system software provides a set of routines and algorithms called the **ColorSync Utilities.** Developers writing device drivers use the ColorSync Utilities to support color matching between devices. You can use the ColorSync Utilities in your application to communicate with a driver and present users with color-matching information—such as a device's color capabilities. For an image that your application prepares, for example, your application can present a print preview dialog box that signifies those colors within the image that the printer cannot accurately reproduce. Your application can also allow users to choose whether and how to match colors in the image with those available on the printer. The ColorSync Utilities are described in *Inside Macintosh: Advanced Color Imaging*.

The **Color Manager** assists Color QuickDraw in mapping your application's color requests to the actual colors available. Most applications never need to call the Color Manager directly. However, for completeness, the routines and data structures of the Color Manager are described in *Inside Macintosh: Advanced Color Imaging*.

Apple has also developed a new, object-based graphics architecture called **QuickDraw GX.** This new architecture provides applications with sophisticated color publishing capabilities. Your application can use QuickDraw GX instead of QuickDraw to create

and draw objects on the screen. Rather than provide a set of drawing commands, as QuickDraw does, QuickDraw GX is built around graphics objects that your application can use as needed.

Your application can also use QuickDraw GX for drawing text. QuickDraw GX provides many sophisticated font and line layout capabilities, such as ligatures, style variations, kerning, and resolution-independent type manipulation.

For printing, QuickDraw GX offers flexible new capabilities to users and an architecture that streamlines development time for developers who write printing drivers. Even if your application uses QuickDraw and the Font Manager instead of QuickDraw GX to create images and text, your application can use the printing capabilities of QuickDraw GX.

See the *Inside Macintosh: QuickDraw GX* suite of books for information about programming with QuickDraw GX imaging technology.

Basic QuickDraw

Contents

About Basic QuickDraw 2-3
The Mathematical Foundations of QuickDraw 2-4
The Coordinate Plane 2-4
Points 2-4
Rectangles 2-5
Regions 2-7
The Black-and-White Drawing Environment: Basic Graphics Ports 2-7
Bitmaps 2-9
The Graphics Port Drawing Area 2-11
Graphics Port Bit Patterns 2-13
The Graphics Pen 2-13
Text in a Graphics Port 2-13
The Limited Colors of a Basic Graphics Port 2-14
Other Fields 2-14
Using Basic QuickDraw 2-14
Initializing Basic QuickDraw 2-16
Creating Basic Graphics Ports 2-16
Setting the Graphics Port 2-18
Switching Between Global and Local Coordinate Systems 2-19
Scrolling the Pixels in the Port Rectangle 2-20
Basic QuickDraw Reference 2-26
Data Structures 2-26
Routines 2-36
Initializing QuickDraw 2-36
Opening and Closing Basic Graphics Ports 2-37
Saving and Restoring Graphics Ports 2-41
Managing Bitmaps, Port Rectangles, and Clipping Regions 2-43
Manipulating Points in Graphics Ports 2-51

Summary of Basic QuickDraw 2-56
Pascal Summary 2-56
Data Types 2-56
Routines 2-57
C Summary 2-58
Data Types 2-58
Functions 2-60
Assembly-Language Summary 2-61
Data Structures 2-61
Global Variables 2-62
Result Codes 2-62

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

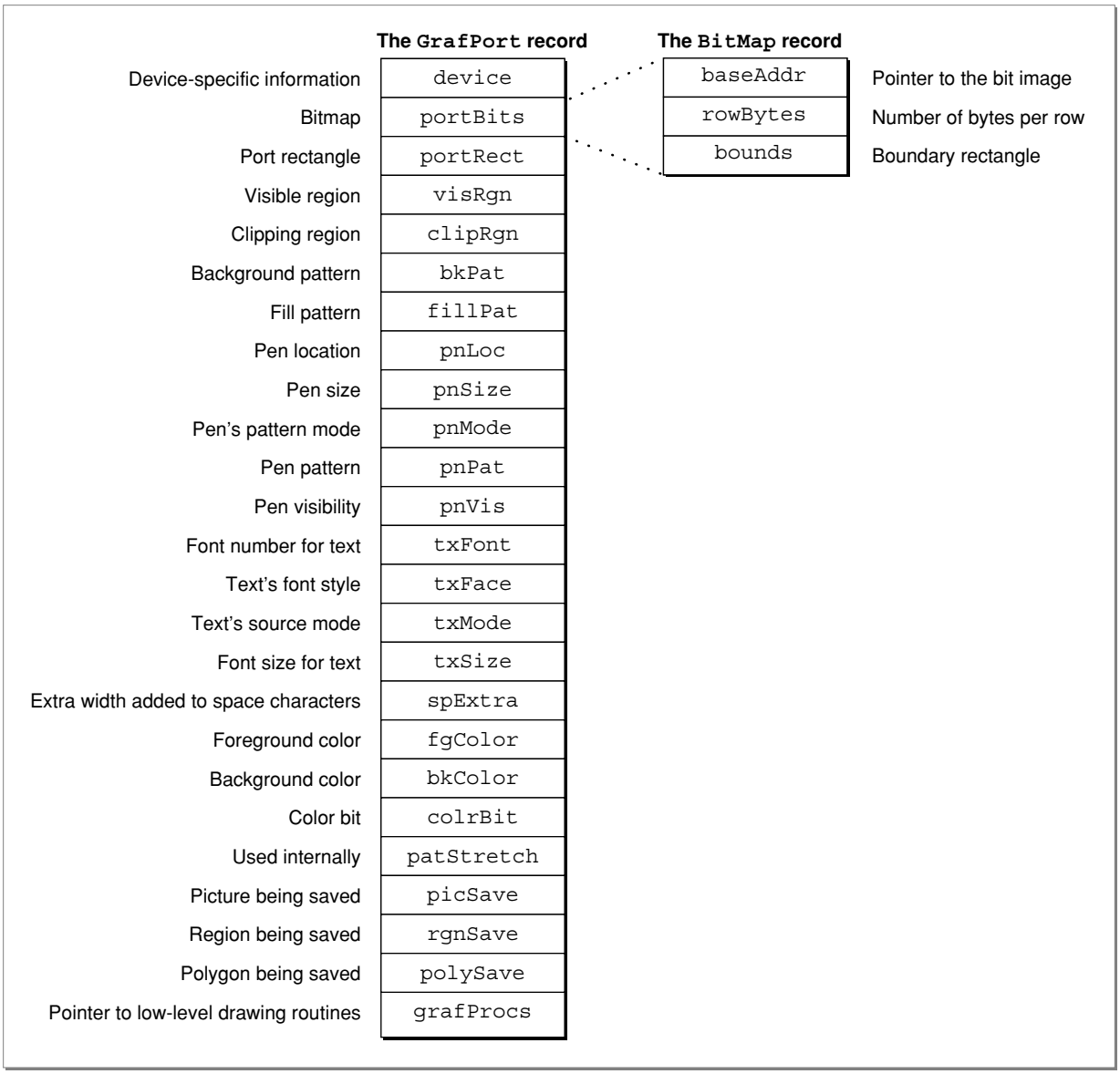
A graphics port is a complete drawing environment that defines where and how graphics operations take place. You can have many graphics ports open at once; each one has its own local coordinate system, drawing pattern, background pattern, pen size and location, font and font style, and bitmap or pixel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

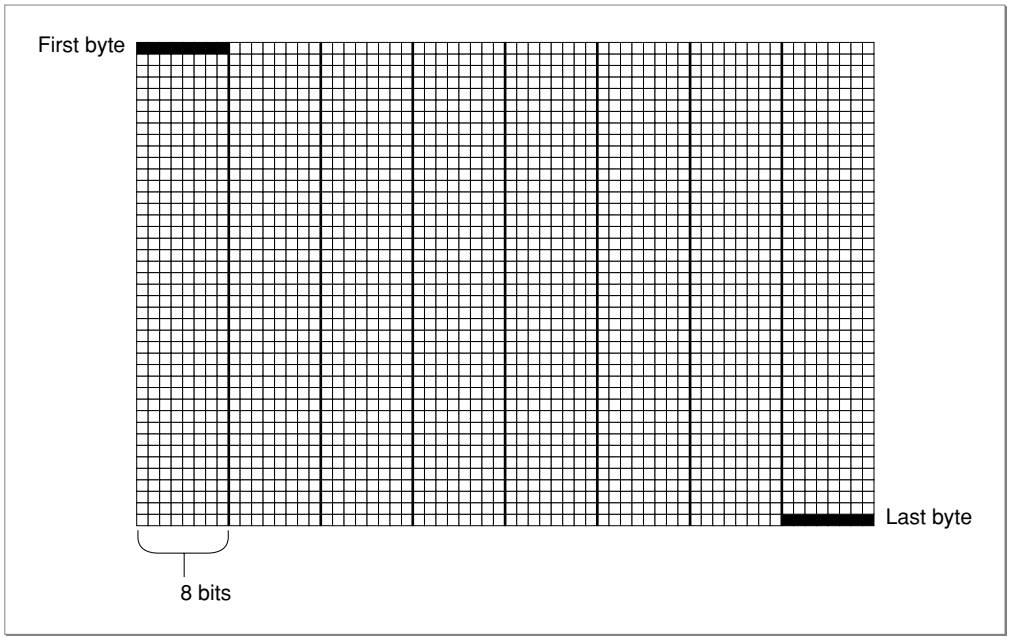


Bitmaps

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



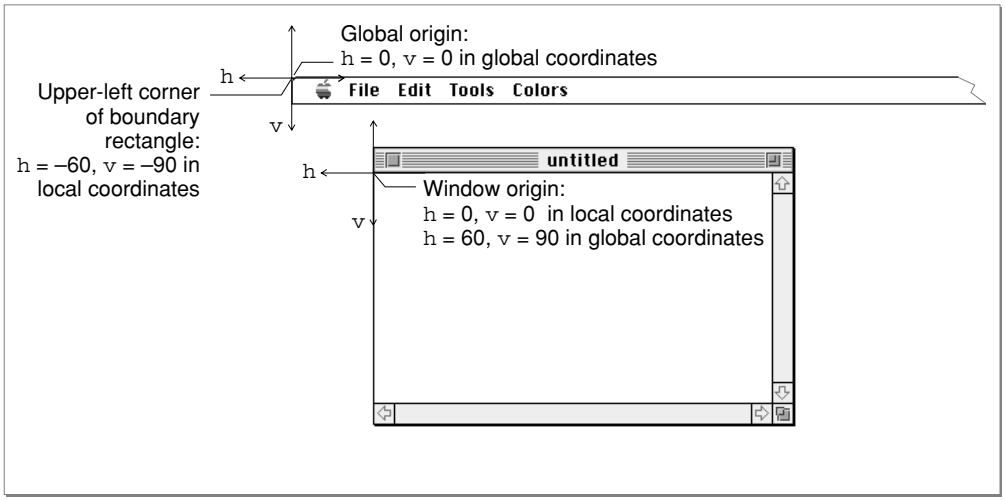
The screen itself is one large visible bit image. On a Macintosh Classic, for example, the screen is a 342-by-512 bit image, with a row width of 64 bytes. These 21,888 bytes of memory are displayed as a matrix of 175,104 pixels on the screen; each bit corresponds to one screen pixel. If a bit's value 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.

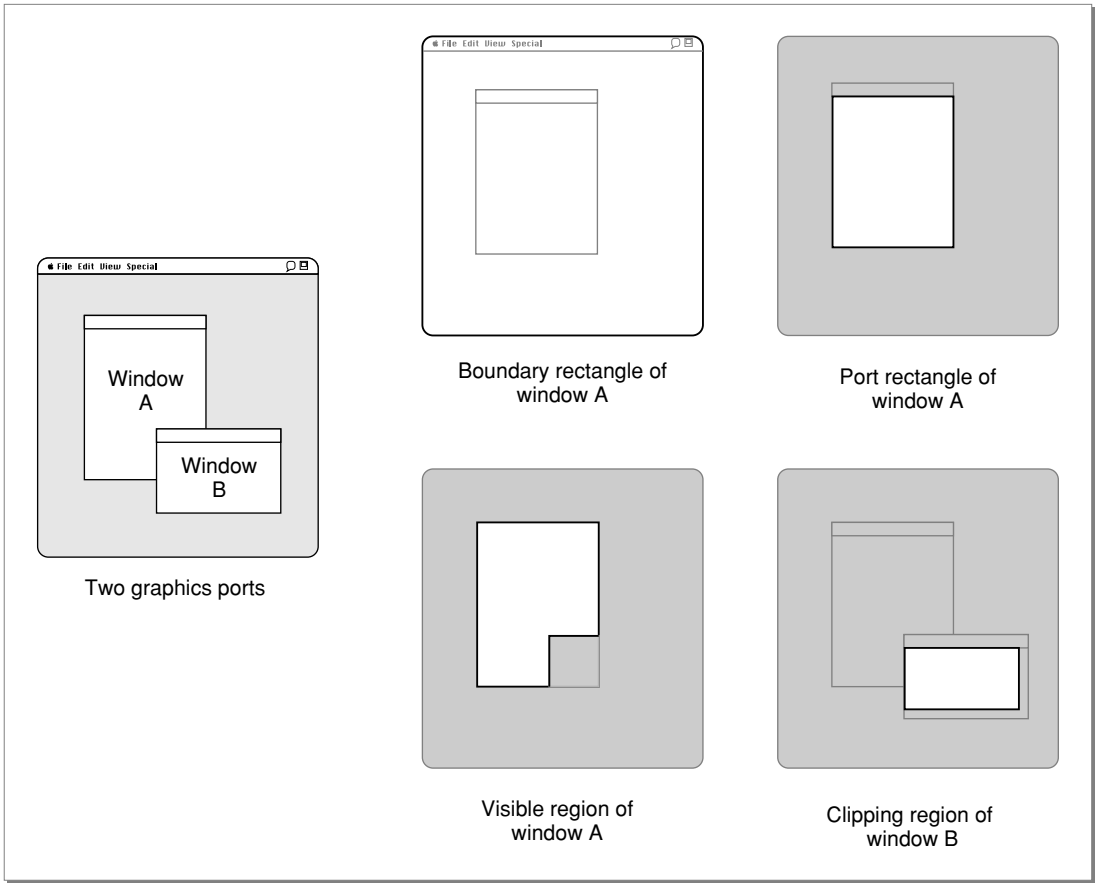
The `visRgn` field designates the visible region of the graphics port. The visible region is the region of the graphics port that’s actually visible on the screen. The visible region is manipulated by the Window Manager. For example, if the user moves one window in front of another, the Window Manager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

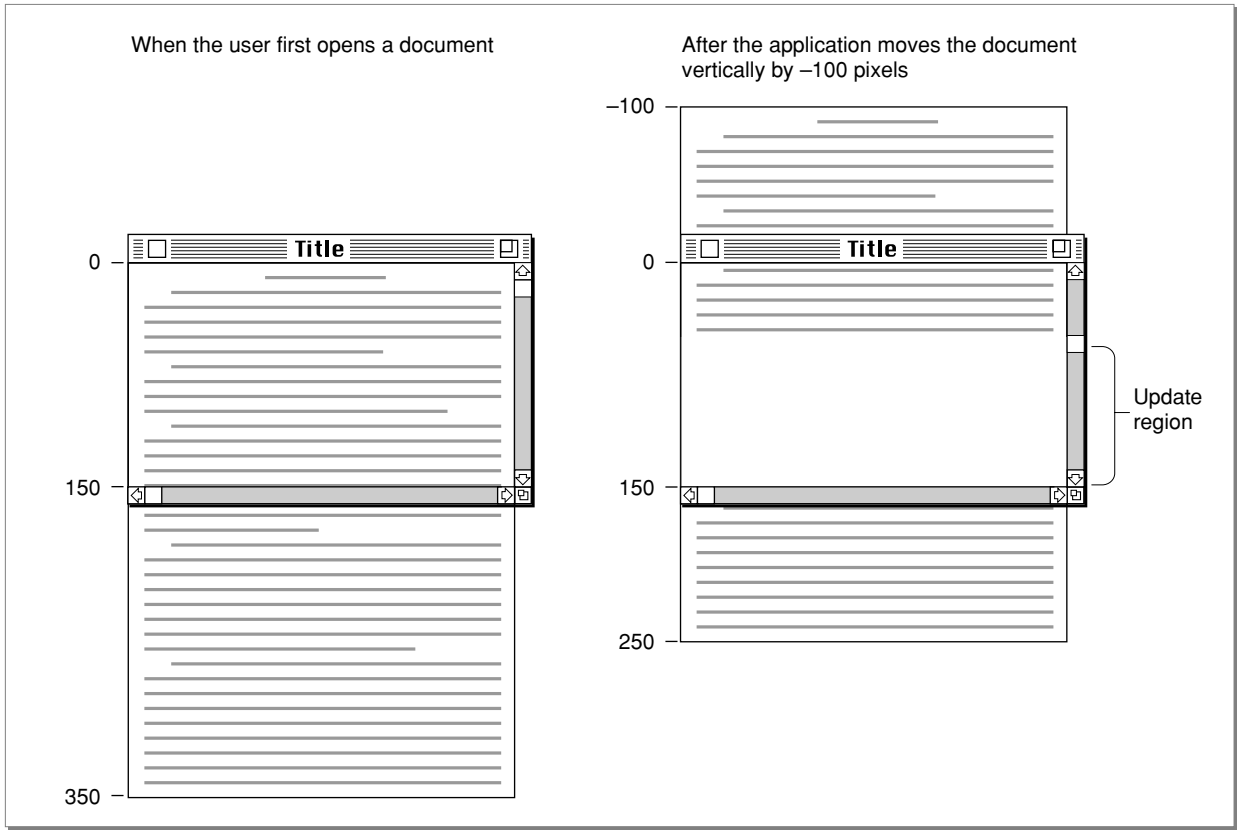
In this example, the content area—that is, the port rectangle—of the window displays 15 lines of text, which amount to 150 pixels.

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.

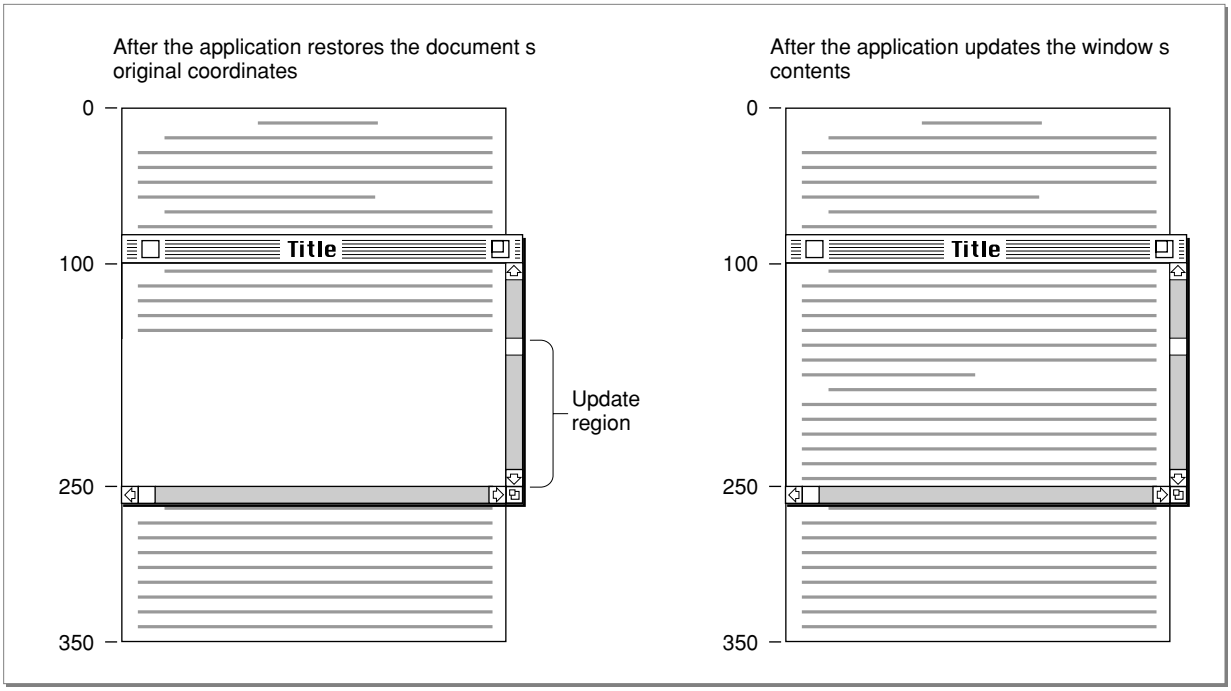
For example, after the user manipulates the vertical scroll bar to move (either up or down) to a location 100 pixels from the top of the document, the application makes the following call:

```
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

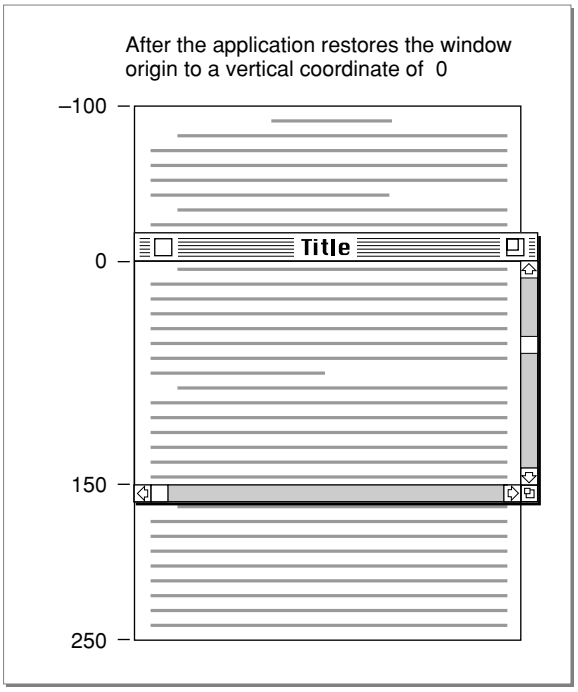


After restoring the document's original coordinates, the application updates the window, as shown on the right side of Figure 2-6. The application draws lines 16 through 24, which it stores in its own document record as beginning at a vertical coordinate of 160 and ending at a vertical coordinate of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

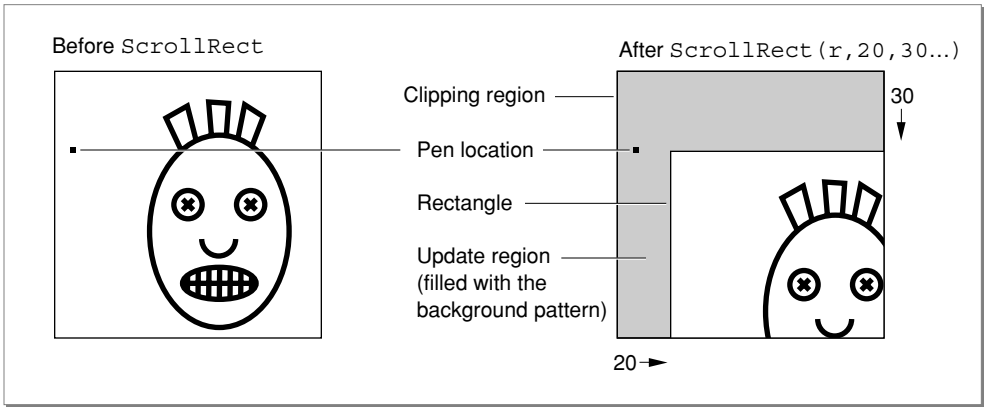
`r`	The rectangle defining the area to be scrolled.
`dh`	The horizontal distance to be scrolled.
`dv`	The vertical distance to be scrolled.
`updateRgn`	A handle to the region of the window that needs to be updated.

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

QuickDraw Drawing

Contents

About QuickDraw Drawing 3-3
- The Graphics Pen 3-4
- Bit Patterns 3-5
- Boolean Transfer Modes With 1-Bit Pixels 3-8
- Lines and Shapes 3-11
  - Defining Lines and Shapes 3-11
  - Framing Shapes 3-12
  - Painting and Filling Shapes 3-12
  - Erasing Shapes 3-12
  - Inverting Shapes 3-13
  - Other Graphic Entities 3-13
- The Eight Basic QuickDraw Colors 3-14

Drawing With QuickDraw 3-16
- Drawing Lines 3-17
- Drawing Rectangles 3-22
- Drawing Ovals, Arcs, and Wedges 3-25
- Drawing Regions and Polygons 3-27
- Performing Calculations and Other Manipulations of Shapes 3-31
- Copying Bits Between Graphics Ports 3-32
- Customizing QuickDraw's Low-Level Routines 3-35

QuickDraw Drawing Reference 3-36
- Data Structures 3-36
- Routines 3-41
  - Managing the Graphics Pen 3-41
  - Changing the Background Bit Pattern 3-48
  - Drawing Lines 3-49
  - Creating and Managing Rectangles 3-52
  - Drawing Rectangles 3-58
  - Drawing Rounded Rectangles 3-63
  - Drawing Ovals 3-68

Drawing Arcs and Wedges 3-71
Creating and Managing Polygons 3-78
Drawing Polygons 3-81
Creating and Managing Regions 3-85
Drawing Regions 3-100
Scaling and Mapping Points, Rectangles, Polygons, and Regions 3-104
Calculating Black-and-White Fills 3-108
Copying Images 3-112
Drawing With the Eight-Color System 3-122
Determining Whether QuickDraw Has Finished Drawing 3-125
Getting Pattern Resources 3-126
Customizing QuickDraw Operations 3-129
Resources 3-140
The Pattern Resource 3-140
The Pattern List Resource 3-141
Summary of QuickDraw Drawing 3-142
Pascal Summary 3-142
Constants 3-142
Data Types 3-144
Routines 3-145
C Summary 3-149
Constants 3-149
Data Types 3-151
Functions 3-152
Assembly-Language Summary 3-157
Data Structures 3-157
Global Variables 3-158

This chapter describes routines common to both basic QuickDraw and Color QuickDraw that you can use to draw lines, rectangles, rounded rectangles, ovals, arcs, wedges, polygons, and regions, and to copy images from one graphics port to another. This chapter also describes the routines that you can use to perform calculations and other manipulations of these shapes—including comparing them and finding their unions and intersections, and moving, shrinking, and expanding them.

Read this chapter to learn how to draw on all models of Macintosh computers. All of the routines described in this chapter depend on your application to create a graphics port drawing environment as described in the previous chapter, "Basic QuickDraw." As noted in this chapter, many of these routines have additional capabilities when performed in the more sophisticated color drawing environments described in the next chapter in this book, "Color QuickDraw." However, if your application does not use color, or uses only a few colors, you may find it unnecessary to create the drawing environment described in the chapter "Color QuickDraw."

This chapter also describes how to copy a bit image from one onscreen graphics port to another onscreen graphics port. To prevent the choppiness that can occur when you build complex images onscreen, your application should use the drawing routines described in this chapter to create complex images in offscreen graphics worlds. Your application can then copy these images to onscreen graphics ports, as described in the chapter "Offscreen Graphics Worlds" in this book.

QuickDraw also supports the creation and manipulation of pictures and text. The chapter "Pictures" in this book describes the routines for drawing pictures. For information about drawing text, see the chapter "QuickDraw Text" in *Inside Macintosh: Text*.

This chapter describes how to

- use the graphics pen
- create, draw, and manipulate the shapes supported by QuickDraw
- draw a copy of a bit image from one graphics port into another graphics port
- use the eight-color system supported by basic QuickDraw
- customize QuickDraw's drawing operations

About QuickDraw Drawing

QuickDraw provides your application with routines for rapidly creating, manipulating, and drawing graphic objects such as lines, arcs, rectangles, ovals, regions, and bitmaps.

These routines extract information from and affect the fields of the current graphics port, without specifically naming it as a parameter. For example, the `Move` procedure moves the graphics pen of the current graphics port, changing the value of its `pnLoc` field, and the `PaintOval` procedure paints an oval using the pattern and pattern mode of the graphics pen for the current graphics port.

The previous chapter, "Basic QuickDraw," describes the basic graphics port. The next chapter, "Color QuickDraw," describes the color graphics port. The routines described in this chapter operate in both types of graphics ports.

Whenever you use QuickDraw, all drawing is performed with the graphics pen, which is described next.

The Graphics Pen

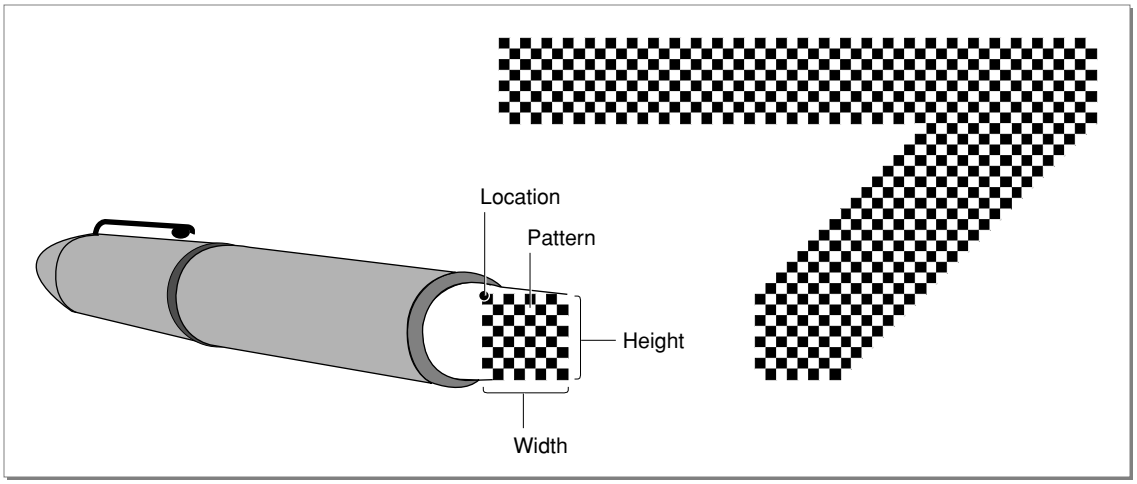
Every graphics port contains one, and only one, graphics pen with which to perform drawing operations. You use this metaphorical pen to draw lines, shapes, and text. Using QuickDraw routines, you can set these five characteristics of the graphics pen for the current graphics port:

- visibility, as stored in the `pnVis` fields of the `GrafPort` and `CGrafPort` records
- size, as stored in the `pnSize` fields of the `GrafPort` and `CGrafPort` records
- location, as stored in the `pnLoc` fields of the `GrafPort` and `CGrafPort` records
- pattern, as stored in the `pnPat` field of the `GrafPort` and `CGrafPort` records
- pattern mode, as stored in the `pnMode` fields of `GrafPort` and `CGrafPort` records

The visibility of the graphics pen simply determines whether the pen draws on the screen. You can use the `HidePen` and `ShowPen` procedures to change the pen's visibility.

The graphics pen is rectangular in shape, and its size (that is, its height and width) are measured in pixels. The default size is a 1-by-1 pixel square, but you can use the `PenSize` procedure to change its shape from a 0-by-0 pixel square to a 32,767-by-32,767 pixel square. If you set either the width or the height to 0, however, the graphics pen does not draw. (Heights or widths of less than 0 are undefined.) Figure 3-1 illustrates a graphics pen of 8 pixels by 8 pixels.

Figure 3-1 A graphics pen



The graphics pen can be located anywhere on the local coordinate plane of the graphics port, and there are no restrictions on the movement or placement of the pen. You can use the `MoveTo` and `Move` procedures to change the pen's location, which is defined by the point that positions the upper-left corner of the pen. You can use the `GetPen` procedure to determine the pen's current location. As shown in Figure 3-1, the pen draws below and to the right of the point specifying its location.

The pattern and pattern mode determine how the bits under the pen are affected when your application draws lines or shapes. A bit pattern is a repeating 8-by-8 bit image, such as that shown in Figure 3-1. You can use the `PenPat` procedure to change the bit pattern for the graphics pen. Bit patterns are described in more detail in the next section. The pattern mode for the graphics pen determines how the bit pattern interacts with the existing bit image according to one of eight Boolean operations, as described in detail in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8. You can use the `PenMode` procedure to change the pattern mode of the graphics pen.

To determine the size, location, pattern, and pattern mode of the graphics pen, you can use the `GetPenState` procedure, which returns a `PenState` record that contains fields for each of these characteristics. If you need to temporarily change these characteristics, you can use the `SetPenState` procedure to restore the graphics pen to the state saved in the record returned by `GetPenState`.

Upon the creation of a graphics port, QuickDraw assigns these initial values to the graphics pen: a size of (1,1), a pattern of all-black pixels, and a pattern mode of `patCopy`. After changing any of these values, you can use the `PenNormal` procedure to return these initial values to the graphics pen.

"Lines and Shapes" beginning on page 3-11 describes how to use the graphics pen to draw lines and shapes.

Bit Patterns

A **bit pattern** is a 64-pixel image, organized as an 8-by-8 pixel square, that defines a repeating design (such as stripes) or a tone (such as gray). The patterns defined in bit patterns are usually black and white, although any two colors can be used on a color screen. Pixel patterns (which are supported only in color graphics ports) define color patterns at any pixel depth. (Pixel patterns are described in the chapter "Color QuickDraw" in this book.) Figure 3-2 shows a typical bit pattern—the one used for the standard gray desktop pattern on most Macintosh computers with black-and-white screens.

Figure 3-2 A bit pattern



You can use bit patterns to draw lines and shapes on the screen. In a basic graphics port, the graphics pen has a pattern specified in the `pnPat` field of its `GrafPort` record. This bit pattern acts like the ink in the pen; the bits in the pattern interact with the pixels in the bitmap according to the pattern mode of the graphics pen. When you use the `FrameRect`, `FrameRoundRect`, `FrameArc`, `FramePoly`, `FrameRgn`, `PaintRect`, `PaintRoundRect`, `PaintArc`, `PaintPoly`, and `PaintRgn` procedures to draw shapes, these procedures draw the shape with the bit pattern specified in the `pnPat` field.

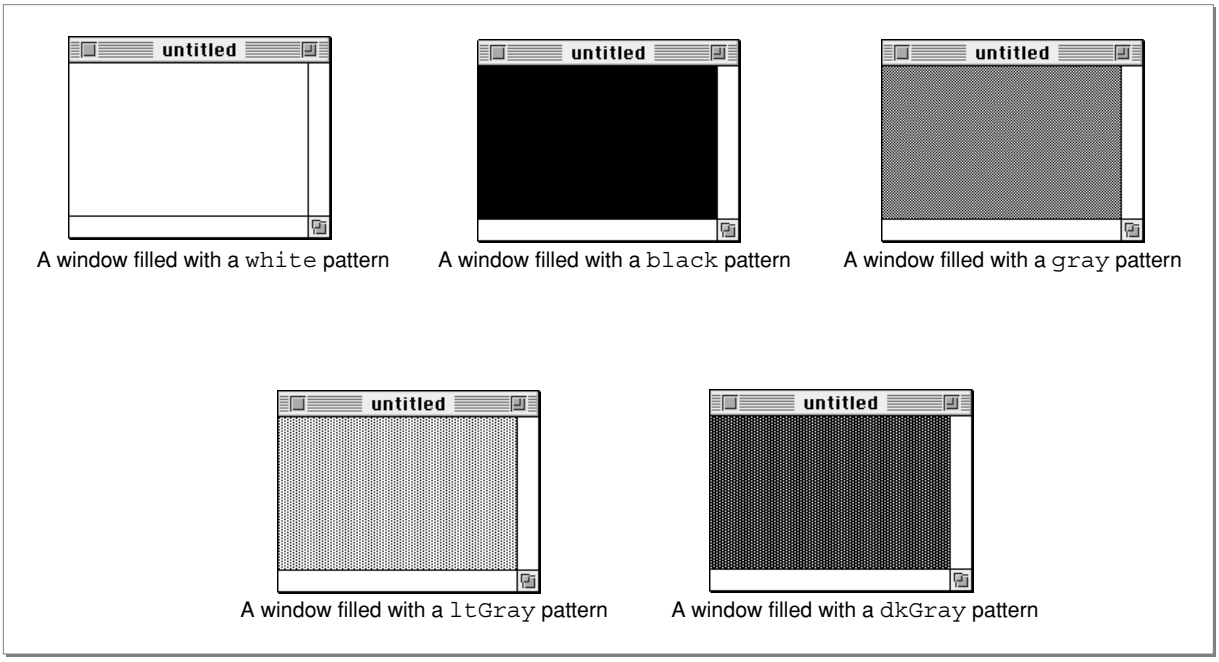
You can use the `FillRect`, `FillRoundRect`, `FillArc`, `FillPoly`, and `FillRgn` procedures to draw shapes with a bit pattern other than that specified in the `pnPat` field of the graphics port. When your application uses one of these procedures, the procedure stores the pattern your application specifies in the `fillPat` field of the `GrafPort` record (or its handle in the `fillPixPat` field of a `CGrafPort` record) and then calls a low-level drawing routine that gets the pattern from that field.

Each graphics port also has a background pattern that's used when an area is erased (such as by using the `EraseRect`, `EraseRoundRect`, `EraseArc`, `ErasePoly`, and `EraseRgn` procedures) and when pixels are scrolled out of an area (such as by using the `ScrollRect` procedure as described in the chapter "Basic QuickDraw"). Every basic graphics port stores a background bit pattern in the `bkPat` field of its `GrafPort` record. (Color graphics ports store a handle to the background pattern in their `bkPixPat` field.)

So that adjacent areas of the same pattern form a continuous, coordinated pattern, all patterns are always drawn relative to the origin of the graphics port.

A basic graphics port supports only bit patterns. Bit patterns are defined in data structures of type `Pattern`, in which each pixel is represented by a single bit. Five such bit patterns are predefined as global variables for your use. These patterns are illustrated in Figure 3-3.

Figure 3-3 Windows filled with the predefined bit patterns



The upper-left window in this figure is filled with the predefined pattern `white`, in which every pixel is white. By default, this is the **background pattern** for a graphics port; that is, this is the pattern displayed when an area is erased or when bits are scrolled out of it. The `white` pattern can also produce useful effects when transferred with an appropriate pattern mode to an existing bit image. (Pattern modes are explained in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8.)

The middle window in the top row of Figure 3-3 is filled with the predefined bit pattern `black`, in which every pixel is black. This is the initial pattern that QuickDraw assigns to the graphics pen.

Figure 3-3 illustrates a window filled with the predefined pattern `gray`, which uses a combination of black and white pixels. As illustrated in this figure, fewer black pixels in the combination produce the predefined pattern `ltGray`, and more black pixels produce the predefined pattern `dkGray`.

These predefined patterns use colored pixels to produce similar effects in color graphics ports, as described in the chapter "Color QuickDraw."

You can create your own bit patterns in your program code, but it's usually simpler and more convenient to store them in resources of type `'PAT '` or `'PAT#'` and to read them in when you need them. The five predefined patterns are available not only through the global variables provided by QuickDraw but also as system resources stored in the system resource file. You can use the `GetPattern` function and the `GetIndPattern` procedure to get patterns stored as resources.

The result of the transfer of a pattern to a bitmap depends on the pattern mode, which is described next.

Boolean Transfer Modes With 1-Bit Pixels

A **Boolean transfer mode** describes an interaction between the pixels that your application draws and the pixels that are already in the destination bitmap—for example, when you draw a patterned line into a graphics port. Black-and-white drawing uses two types of Boolean transfer modes:

- **source modes** for copying bit images or drawing text
- **pattern modes** for drawing lines and shapes

Color QuickDraw uses Boolean transfer mode differently than basic QuickDraw. Color QuickDraw also has transfer modes that perform arithmetic operations on the red, green, and blue values of color pixels. Using transfer modes with Color QuickDraw is described in the chapter "Color QuickDraw" in this book.

Your application uses source modes when using `CopyBits` procedure (described in "Copying Bits Between Graphics Ports" beginning on page 3-32) and the `CopyDeepMask` procedure (described in the chapter "Color QuickDraw").

Your application uses pattern modes to transfer patterns to lines and shapes. The `penMode` field of a graphics port stores the pattern mode for the graphics pen. You use the pattern mode to draw lines, rectangles, rounded rectangles, ovals, arcs, wedges, polygons, and regions, as follows:

- When you use a procedure like `LineTo`, `FrameRect`, or `FrameOval`, the procedure draws the lines of your shape with the pattern specified in the `pnPat` field of the graphics port, but the procedure transfers that pattern into the graphics port by using the pattern mode specified in the `pnMode` field of the current graphics port.
- When you use a procedure like `PaintRect` or `PaintOval`, the procedure draws your shape with the pattern specified in the `pnPat` field by transferring the pattern with the pattern mode specified in the `pnMode` field.
- When you use a procedure like `FillRect` or `FillOval`, the procedure draws your shape with the pattern you request and uses the `patCopy` pattern mode (which copies your requested pattern directly into the shape).

You use the source mode when using the `CopyBits` procedure to copy a bit image from one graphics port to another and when drawing text using the QuickDraw routines described in the chapter "QuickDraw Text" in *Inside Macintosh: Text*. (The source mode for text is stored in the `textMode` field of a graphics port.)

For both pattern and source modes there are four Boolean operations: COPY, OR, XOR (for exclusive-or), and BIC (for bit clear). Each of these operations has an inverse variant in which the pattern or source is inverted before the transfer, so in fact there are eight operations in all.

The eight operations in the pattern and source modes have names defined as constants. Their effects on 1-bit destination pixels are summarized in Table 3-1. (See the chapter "Color QuickDraw" for information about the effects of these operations on colored pixels—that is, those with a pixel depth of more than 1 pixel.)

Table 3-1 Effect of Boolean transfer modes on 1-bit pixels

Pattern mode	Source mode	Action on destination pixel	
		If pattern or source pixel is black	If pattern or source pixel is white
`patCopy`	`srcCopy`	Force black	Force white
`notPatCopy`	`notSrcCopy`	Force white	Force black
`patOr`	`srcOr`	Force black	Leave alone
`notPatOr`	`notSrcOr`	Leave alone	Force black
`patXor`	`srcXor`	Invert	Leave alone
`notPatXor`	`notSrcXor`	Leave alone	Invert
`patBic`	`srcBic`	Force white	Leave alone
`notPatBic`	`notSrcBic`	Leave alone	Force white

The COPY operations completely replace the pixels in the destination bitmap with either the pixels in the pattern (for the `patCopy` mode) or the pixels in the source bitmap (for the `srcCopy` mode). The inverse COPY operations completely replace the pixels in the destination bitmap with a "photographic negative" of the pattern (for the `notPatCopy` mode) or the source bitmap (for the `notSrcCopy` mode).

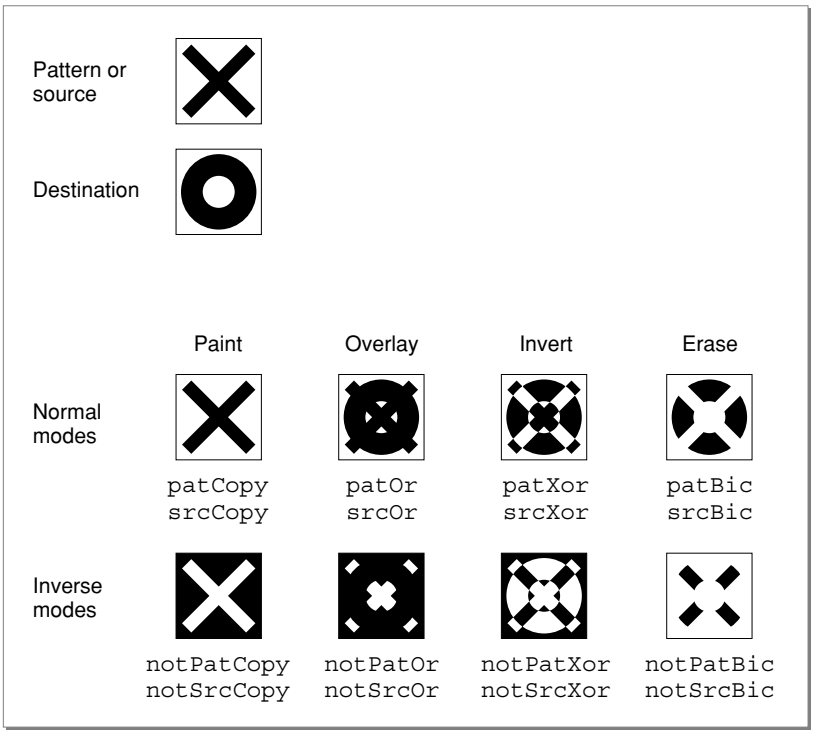
The OR operations add the black pixels from either the pattern (for the `patOr` mode) or the source bitmap (for the `srcOr` mode) to the destination bitmap. The inverse OR operations (`notPatOr` and `notSrcOr` modes) take a "photographic negative" of the pattern or the source bitmap, and then add the black pixels from this negative to the destination bitmap.

The XOR operations (`patXor` and `srcXor` modes) invert the pixels in the destination bitmap that correspond to black pixels in the pattern or source bitmap. The inverse XOR operations (`notPatXor` and `notSrcXor` modes) invert the pixels in the destination bitmap that correspond to white pixels in the pattern or source bitmap.

The BIC operations (`patBic` and `srcBic` modes) turn pixels in the destination bitmap white when they correspond to black pixels in the pattern or source bitmap. The inverse BIC operations (`notPatBic` and `notSrcBic` modes) turn pixels in the destination bitmap white when they correspond to white pixels in the pattern or source bitmap.

These actions are illustrated in Figure 3-4, where a black X is transferred to a destination bitmap consisting of a black O.

Figure 3-4 Examples of Boolean transfer modes



On computers running System 7, you can add dithering to any source mode by adding the following constant or the value it represents to the source mode:

```
CONST ditherCopy  = 64;
```

Dithering mixes existing colors to create the effect of additional colors on indexed devices. It also improves images that you shrink or that you copy from a direct pixel device to an indexed device. Using dithering even when shrinking 1-bit images between basic graphics ports can produce much better representations of the original images. The `CopyBits` procedure always dithers images when shrinking them between pixel maps on direct devices. Dithering is explained in the chapter "Color QuickDraw."

The next section describes how your application uses pattern modes to transfer patterns to lines and shapes.

Lines and Shapes

As explained in the chapter "Basic QuickDraw," rectangles and regions are mathematical models that QuickDraw defines as data types. However, they also can be graphic elements that appear on the screen. A rectangle, for example, can mathematically define a visible area, but it can also be an object to draw.

Defining Lines and Shapes

You use two points to define a line. Using the `LineTo` and `Line` procedures, you can draw lines onscreen using the size, pattern, and pattern mode of the graphics pen for the current graphics port. You can also define a rectangle with two points (the upper-left and lower-right corners of the rectangle) or with four boundary coordinates (one for each side of the rectangle). Using the `FrameRect` procedure, you can draw rectangles that are framed by lines rendered with the size, pattern, and pattern mode of the graphics pen.

You use rectangles to define ovals and rounded rectangles. Rectangles used to define other shapes are called **bounding rectangles.** The lines of bounding rectangles completely enclose the shapes they bound; in other words, no pixels from these shapes lie outside the infinitely thin lines of the bounding rectangles.

Ovals are circular or elliptical shapes defined by the height and width of their bounding rectangles, and rounded rectangles are rectangles with rounded corners defined by the width and height of the ovals forming their corners. Using the `FrameOval` and `FrameRoundRect` procedures, you can draw, respectively, framed ovals and framed rounded rectangles.

You can use rectangles to define ovals that, in turn, you can use to define arcs and wedges. An arc is a portion of an oval's circumference bounded by a pair of radii. A wedge is a pie-shaped segment of an oval. The wedge starts at the center of the oval, is bounded by a pair of radii, and extends to the oval's circumference. You use the `FrameArc` procedure to draw a framed arc, and you use the `PaintArc` or `FillArc` procedure to draw a wedge.

You use lines to define a polygon. First, however, you must call the `OpenPoly` function and then some number of `LineTo` procedures to create lines from the first vertex of the polygon to the second, from the second to the third, and so on, until you've created a line to the last vertex. You then use the `ClosePoly` procedure, which completes the figure by drawing a connecting line from the last vertex back to the first. After defining a polygon in this way, you can draw a framed outline of it using the `FramePoly` procedure.

To define a region, you can use any set of lines or shapes, including other regions, so long as the region's outline consists of one or more closed loops. First, however, you must call the `NewRgn` function and `OpenRgn` procedure. You then use line-, shape-, or region-drawing commands to define the region, which can be concave or convex, can consist of one connected area or many separate ones, and can even have holes in the middle. When you are finished collecting commands to define the outline of the region, you use the `CloseRgn` procedure. You can then draw a framed outline of the region using the `FrameRgn` procedure.

Framing Shapes

Using the `FrameRect`, `FrameOval`, `FrameRoundRect`, `FrameArc`, `FramePoly`, or `FrameRgn` procedure to **frame** a shape draws just its outline, using the size, pattern, and pattern mode of the graphics pen for the current graphics port. The interior of the shape is unaffected, allowing previously existing pixels in the bit image to show through.

Painting and Filling Shapes

Using the `PaintRect`, `PaintOval`, `PaintRoundRect`, `PaintArc`, `PaintPoly`, or `PaintRgn` procedure to **paint** a shape draws both its outline and its interior with the pattern of the graphics pen, using the pattern mode of the graphics pen.

Using the `FillRect`, `FillOval`, `FillRoundRect`, `FillArc`, `FillPoly`, or `FillRgn` procedure to **fill** a shape draws both its outline and its interior with any pattern you specify. The procedure transfers the pattern with the `patCopy` pattern mode, which directly copies your requested pattern into the shape.

Erasing Shapes

Using the `EraseRect`, `EraseOval`, `EraseRoundRect`, `EraseArc`, `ErasePoly`, or `EraseRgn` procedure to **erase** a shape draws both its outline and its interior with the background pattern for the current graphics port. The background pattern is typically solid white on a black-and-white monitor or a solid background color on a color monitor. Making the shape blend into the background pattern of the graphics port effectively erases the shape.

Inverting Shapes

Using the `InvertRect`, `InvertOval`, `InvertRoundRect`, `InvertArc`, `InvertPoly`, or `InvertRgn` procedure to **invert** a shape reverses the colors of all pixels within its boundary. On a black-and-white monitor, this changes all the black pixels in the shape to white and changes all the white pixels to black.

The inversion procedures were designed for 1-bit images in basic graphics ports. These procedures operate on color pixels in color graphics ports, but the results are predictable only with direct devices or 1-bit pixel maps.

For indexed pixels, Color QuickDraw performs the inversion on the pixel indexes. The results depend entirely on the contents of the video device's color lookup table (CLUT). (The CLUT is described in the chapter "Color QuickDraw.")

The eight colors used in basic QuickDraw are stored in a color table represented by the global variable `QDColors`. To display those eight basic QuickDraw colors on an indexed device, Color QuickDraw uses the Color Manager to obtain indexes to the colors in the CLUT that best map to the colors in the `QDColors` color table. Because the index, not the color value, is inverted, the results are again unpredictable. (The eight-color system is described in "The Eight Basic QuickDraw Colors" beginning on page 3-14.)

Inversion works better for direct devices. Inverting a pure green, for example, that has red, green, and blue component values of $0000, $FFFF, and $0000 results in magenta, which has component values of $FFFF, $0000, and $FFFF.

Other Graphic Entities

"Drawing With QuickDraw" beginning on page 3-16 provides an introduction to creating and drawing lines, rectangles, rounded rectangles, ovals, arcs, wedges, polygons, and regions. You can also use QuickDraw routines to draw pictures, cursors, icons, and text.

A QuickDraw picture is the recorded transcription of a sequence of drawing operations that can be played back with the `DrawPicture` procedure. See the chapter "Pictures" for information about creating and displaying QuickDraw pictures.

A cursor is a 16-by-16 pixel image that maps the user's movement of the mouse to relative locations on the screen. An icon is an image (usually 32 by 32 or 16 by 16 pixels) that represents an object, a concept, or a message. For example, the Finder uses icons to represent files and disks. Cursors and icons are stored as resources. See the chapter "Cursor Utilities" for information about drawing cursors. See the chapter "Icon Utilities" in *Inside Macintosh: More Macintosh Toolbox* for information about drawing icons.

See the chapter "QuickDraw Text" in *Inside Macintosh: Text* for information about using QuickDraw routines to draw text.

The Eight Basic QuickDraw Colors

On a color screen, you can draw with colors, even when you are using a basic graphics port. Although basic QuickDraw graphics routines were designed for black-and-white drawing, they also support the **eight-color system** that basic QuickDraw predefines for display on color screens and color printers. Because Color QuickDraw also supports this system, it is compatible across all Macintosh platforms. (This section describes the rudimentary color capabilities included in basic QuickDraw. See the next chapter, "Color QuickDraw," for information about more sophisticated color use in your application.)

A pair of fields in a graphics port, `fgColor` and `bkColor`, specify a foreground and background color. The **foreground color** is the color used for bit patterns and for the graphics pen when drawing. By default, the foreground color is black. The **background color** is the color of the pixels in the bitmap wherever no drawing has taken place. By default, the background color is white. However, you can use the `ForeColor` and `BackColor` procedures to change these fields. (When printing, however, use the `ColorBit` procedure to set the foreground color.) For example, on a color screen the following lines of code draw a red rectangle against a blue background.

```
BackColor(blueColor);   {make a blue background}
ForeColor(redColor);    {draw with red ink}
PenMode(patCopy);       {when drawing, replace background color }
                        { with ink's color}
PaintRect(20,20,80,80); {create and paint the red rectangle}
```

If you use the `OpenCPicture` or `OpenPicture` function to include this code in a picture definition, these colors are stored in the picture. However, basic QuickDraw cannot store these colors in a bitmap. See the chapter "Pictures" in this book for more information about defining and drawing pictures.

The basic QuickDraw color values consist of 1 bit for normal black-and-white drawing (black on white), 1 bit for inverted black-and-white drawing (white on black), 3 bits for the additive primary colors (red, green, blue) used in video display, and 4 bits for the subtractive primary colors (cyan, magenta, yellow, black) used in printing. QuickDraw includes a set of predefined constants for those standard colors:

```
CONST
  whiteColor   = 30;
  blackColor   = 33
  yellowColor  = 69;
  magentaColor = 137;
  redColor     = 205;
  cyanColor    = 273;
  greenColor   = 341;
  blueColor    = 409;
```

These are the only colors available in basic QuickDraw (or with Color QuickDraw drawing into a basic graphics port). When you specify these colors on a Macintosh computer with Color QuickDraw, Color QuickDraw draws these colors if the user has set the screen to a color mode.

These eight color values are based on a planar model: each bit position corresponds to a different color plane, and the value of each bit indicates whether a particular color plane should be activated. (The term *color plane* refers to a logical plane, rather than a physical plane.) The individual color planes combine to produce the full-color image.

There are three advantages to using basic QuickDraw's color system:

- It is available across all platforms, so you don't have to check for the presence of Color QuickDraw.
- It is much simpler to use than Color QuickDraw.
- It works well on an ImageWriter printer with a color ribbon.

The main disadvantage is that basic QuickDraw is limited to eight predefined colors. Another problem is that, if the graphics port in which you are working happens to be a color graphics port, then the two color systems may clash. For example, saving the current foreground color (from the `fgColor` field of the color graphics port) and then later restoring it with the `ForeColor` procedure doesn't work: the original content of the `fgColor` field is an index value for a color graphics port using indexed colors. This index value is not what basic QuickDraw's `ForeColor` procedure expects as a parameter.

In System 7, these Color QuickDraw routines are available to basic QuickDraw: `RGBForeColor`, `RGBBackColor`, `GetForeColor`, and `GetBackColor`. Described in the next chapter, "Color QuickDraw," these routines can also assist you in manipulating the eight-color system of basic QuickDraw. When running on a System 7 computer, your application should use `GetForeColor` and `GetBackColor` to determine the foreground color and background color instead of checking the `fgColor` and `bkColor` fields of the `GrafPort` record.

The next section provides an introduction to creating and drawing lines and shapes. Without using a color graphics port, you can use the `ForeColor` or `RGBForeColor` procedure on a color screen to draw these lines and shapes in color, against the background color you set with the `BackColor` or `RGBBackColor` procedure.

Drawing With QuickDraw

You can use QuickDraw's basic drawing routines to

- draw lines of various thicknesses and in various patterns
- draw rectangles, rounded rectangles, ovals, arcs, wedges, polygons, and regions in various patterns
- draw lines and shapes in any of eight predefined colors, against a background of any of these eight predefined colors
- perform calculations on and manipulate rectangles and regions
- copy bits from the bit image in the bitmap of one graphics port into the bitmap of another graphics port
- customize QuickDraw's drawing behavior

System software uses QuickDraw's drawing routines to implement the Macintosh user interface. The next several sections provide an introduction to these routines, which your application can use to create complex onscreen images.

To draw lines, your application

- moves the graphics pen to a location within its graphics port
- draws a line to a different coordinate

To draw rectangles, rounded rectangles, ovals, arcs, and wedges, your application generally

- defines the outline of the shape in the local coordinates of the graphics port
- frames the shape's outline to draw it
- transfers patterns to the outline and interior of the shape to paint or fill it

To draw regions and polygons, your application

- uses an open routine to start building the shape
- calls drawing routines to build the shape
- uses a close routine to stop collecting drawing routines for the shape
- frames the shape's outline to draw it
- transfers patterns to the outline and interior of the shape to paint or fill it

These tasks are explained in greater detail in the rest of this chapter.

Before using QuickDraw's drawing routines, you must initialize QuickDraw with the `InitGraf` procedure, as explained in the chapter "Basic QuickDraw."

The routines described in this chapter are available on all models of Macintosh computers. However, all nonwhite colors that you specify with the `ForeColor` and `BackColor` procedures are displayed as black on a black-and-white screen. Before using the `ForeColor` and `BackColor` procedures to display colors in a basic graphics port, you can use the `DeviceLoop` procedure, which is described in the chapter "Graphics Device," to determine the color characteristics of the current screen.

Drawing Lines

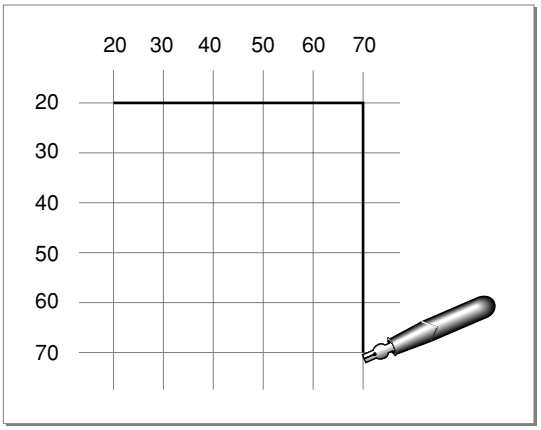
A line is defined by two points: the current location of the graphics pen and its destination. The graphics pen draws below and to the right of the defining points. As described in "The Graphics Pen" on page 3-4, the pen draws the line between its defining points with the size, pattern, and pattern mode stored in the current graphics port.

You specify where to begin drawing a line by using the `MoveTo` or `Move` procedure to place the graphics pen at some point in the window's local coordinate system. Then you call the `LineTo` or `Line` procedure to draw a line from there to another point. Take, for example, the following lines of code:

```
MoveTo(20,20);
LineTo(70,20);
LineTo(70,70);
```

The `MoveTo` procedure moves the graphics pen to a point with a horizontal coordinate of 20 and a vertical coordinate of 20 (in the local coordinate system of the graphics port). The first call to the `LineTo` procedure draws a line from that position to the point with a horizontal coordinate of 70 and a vertical coordinate of 20. The second call to the `LineTo` procedure draws a line from the pen's new position to the point with a horizontal coordinate of 70 and a vertical coordinate of 70, as shown in Figure 3-5.

Figure 3-5 Using the `LineTo` procedure



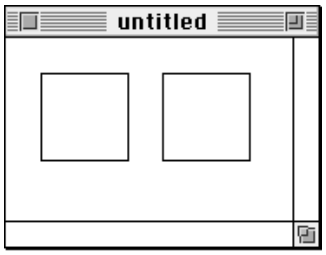
Listing 3-1 illustrates how to use the `LineTo` procedure to draw the four sides of a square, which is shown on the left side of Figure 3-6. In Figure 3-6, the current graphics port is the window "untitled."

Listing 3-1 Drawing lines with the `LineTo` and `Line` procedures

```
PROCEDURE MyDrawLines;
BEGIN
   MoveTo(20,20);
   LineTo(70,20);
   LineTo(70,70);
   LineTo(20,70);
   LineTo(20,20);

   Move(70,0);
   Line(50,0);
   Line(0,50);
   Line(-50,0);
   Line(0,-50);
END;
```

Figure 3-6 Drawing lines

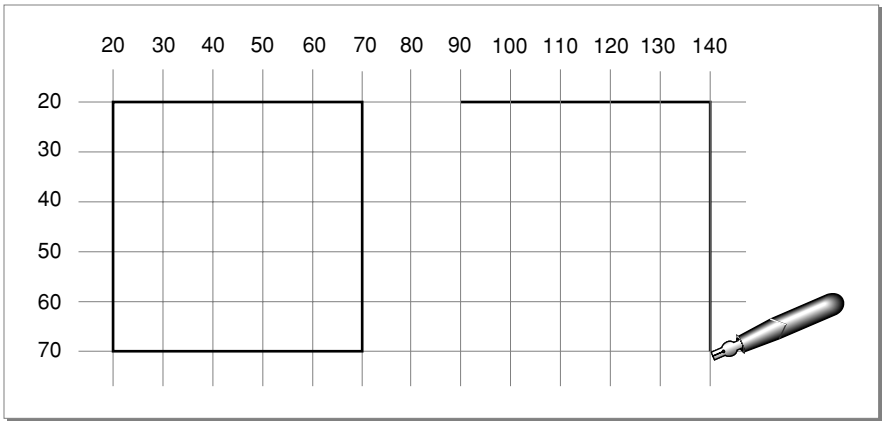


The `MoveTo` and `LineTo` procedures require you to specify a point in the local coordinate system of the current graphics port. These procedures then transfer the graphics pen to that specific location. As alternatives to using the `MoveTo` and `LineTo` procedures, you can use the `Move` and `Line` procedures, which require you to pass relative horizontal and vertical distances to move the pen from its current location. The square on the right side of Figure 3-6 is drawn using the `Move` and `Line` procedures.

The final call to `LineTo` in Listing 3-1 moves the graphics pen to the point with a horizontal coordinate of 20 and a vertical coordinate of 20. Listing 3-1 then uses the `Move` procedure to move the graphics pen a horizontal distance of 70 points—that is, to the point with a horizontal coordinate of 90. The first call to the `Line` procedure draws a

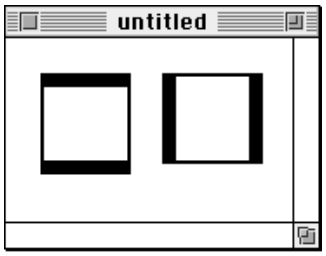
horizontal line 50 pixels long—that is, to the point with a horizontal coordinate of 140 and a vertical coordinate of 20. Starting from there, the second call to `Line` draws a vertical line 50 pixels long—that is, to the point with a horizontal coordinate of 140 and a vertical coordinate of 70, as shown in Figure 3-7.

Figure 3-7 Using the `LineTo` and `Line` procedures



In Figure 3-6, the lines are drawn using the default pen size (1,1), giving the line a vertical depth of one pixel and a horizontal width of one pixel. You can use the `PenSize` procedure to change the width and height of the graphics pen so that it draws thicker lines, as shown in Figure 3-8.

Figure 3-8 Resizing the pen



The square on the left side of Figure 3-8 is drawn with a pen that has a width of two pixels and a height of eight pixels. The square on the right side of this figure is drawn with a pen that has a width of eight pixels and a height of two pixels. Listing 3-2 shows the code that draws these squares.

Listing 3-2 Using the `PenSize` procedure

```
PROCEDURE MyResizePens;
BEGIN
   PenSize(2,8);
   MoveTo(20,20);
   LineTo(70,20); LineTo(70,70); LineTo(20,70); LineTo(20,20);
   PenSize(8,2);
   Move(70,0);
   Line(50,0); Line(0,50); Line(-50,0); Line(0,-50);
   PenNormal;
END;
```

At the end of Listing 3-2, the `PenNormal` procedure is used to restore the graphics pen to its default size, pattern, and pattern mode.

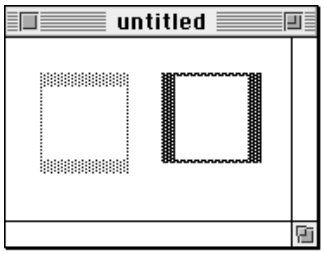
The default pattern for the graphics pen consists of all black pixels. However, you can use the `PenPat` procedure to change the pen's pattern. When you use the `PenPat` procedure, you can pass it any one of the predefined global variables listed in to specify the bit pattern for the graphics pen.

Table 3-2 The global variables for five predefined bit patterns

Global variable	Result
`black`	All-black pattern
`dkGray`	75% gray pattern
`gray`	50% gray pattern
`ltGray`	25% gray pattern
`white`	All-white pattern

In Figure 3-9, the pen pattern for the square on the left has changed to `ltGray`; the pen pattern for the square on the right has changed to `dkGray`.

Figure 3-9 Changing the pen pattern



Listing 3-3 shows the code that produces these squares.

Listing 3-3 Using the `PenPat` procedure to change the pattern of the graphics pen

```
PROCEDURE MyRepatternPens;
BEGIN
   PenSize(2,8);
   PenPat(ltGray);
   MoveTo(20,20);
   LineTo(70,20); LineTo(70,70); LineTo(20,70); LineTo(20,20);

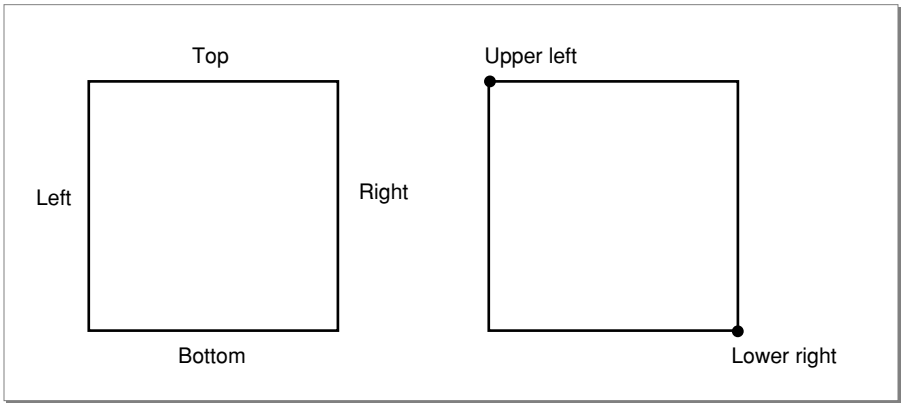
   PenSize(8,2);
   PenPat(dkGray);
   Move(70,0);
   Line(50,0); Line(0,50); Line(-50,0); Line(0,-50);
   PenNormal;
END;
```

QuickDraw provides methods for drawing squares and rectangles that are easier than drawing each side individually as a line. The next section describes how to draw rectangles.

Drawing Rectangles

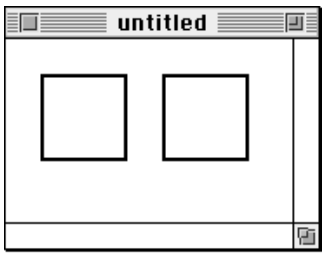
As explained in the chapter "Basic QuickDraw," rectangles are mathematical entities. There are two ways to specify a rectangle: by its four boundary coordinates, as shown in the left rectangle in Figure 3-10, or by its upper-left and lower-right points, as shown in the right rectangle.

Figure 3-10 Two ways to specify a rectangle



However, specifying a rectangle does not draw one. Because the border of a rectangle is infinitely thin, it can have no direct representation on the screen until you use the `FrameRect` procedure to draw its outline, or you can use the `PaintRect` or `FillRect` procedure to draw its outline and its interior with a pattern. Figure 3-11 illustrates two rectangles that are drawn with the `FrameRect` procedure.

Figure 3-11 Drawing rectangles



Listing 3-4 shows the code that draws the rectangles in Figure 3-11. This listing uses the `PenSize` procedure to assign a size of (2,2) to the graphics pen. Then the code assigns four boundary coordinates to the rectangle on the left side of this figure, and it calls `FrameRect` to use the graphics pen to draw the rectangle's outline.

Listing 3-4 Using the `FrameRect` procedure to draw rectangles

```
PROCEDURE MyDrawRects;
   VAR
      firstRect, secondRect: Rect;
BEGIN
   PenSize(2,2);

   firstRect.top := 20;
   firstRect.left := 20;
   firstRect.bottom := 70;
   firstRect.right := 70;
   FrameRect(firstRect);

   SetRect(secondRect,90,20,140,70);
   FrameRect(secondRect);

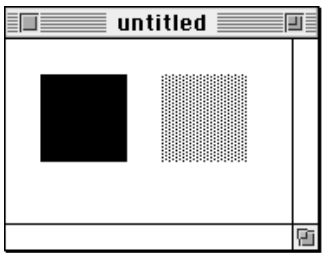
   PenNormal;
END;
```

To shorten code text, Listing 3-4 uses the `SetRect` procedure to define the rectangle on the right side of Figure 3-11. Again, `FrameRect` is used to draw an outline around the rectangle. Notice that while a `Rect` record lists the fields for a rectangle's boundaries in the order `top`, `left`, `bottom`, and `right`, you pass these boundaries as parameters to the `SetRect` procedure in the order `left`, `top`, `right`, and `bottom`.

Remember that the graphics pen hangs to the right of and below its location point; therefore, the lower-right corner of the two-pixel outline around the rectangle on the right side of Figure 3-11 lies at the point with a horizontal coordinate of 142 and a vertical coordinate of 72.

Figure 3-12 illustrates painted and filled rectangles. Listing 3-5 shows the code that creates these images.

Figure 3-12 Painting and filling rectangles



Listing 3-5 uses the `PaintRect` procedure to draw the outline and the interior of the rectangle on the left side of Figure 3-12 with the pattern of the graphics pen, according to the pattern mode of the graphics pen. Because Listing 3-5 calls the `PenNormal` procedure immediately before calling `PaintRect`, the graphics pen has its default characteristics: a pattern of all-black pixels and the `patCopy` pattern mode, which changes all of the pixels in the destination to the pen's pattern.

Listing 3-5 Using the `PaintRect` and `FillRect` procedures

```
PROCEDURE MyPaintAndFillRects;
   VAR
   firstRect, secondRect: Rect;
BEGIN
   PenNormal;
   SetRect(firstRect,20,20,70,70);
   PaintRect(firstRect);
   SetRect(secondRect,20,90,70,140);
   FillRect(secondRect,ltGray);
END;
```

The `PaintRect` procedure always uses the pattern and pattern mode of the graphics pen when drawing a rectangle. If you want to use a pattern other than that of the graphics pen, you can use the `FillRect` procedure. The `FillRect` procedure, however, *always* uses the `patCopy` pattern mode. Listing 3-5 uses the `FillRect` procedure to draw the outline and the interior of the rectangle on the right side of Figure 3-12 with a light gray pattern.

Note
Neither the `PaintRect` nor `FillRect` procedure changes the location of the graphics pen. ◆

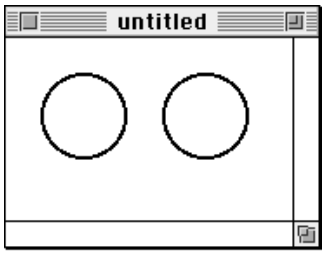
If the application that draws the rectangles in Figure 3-12 uses the `EraseRect` procedure to erase them both, then they would be filled with the background pattern specified by the `bkPat` field of the current graphics port. If the application uses the `InvertRect` procedure to invert the rectangles, then the black pixels in each would become white and the white pixels would become black.

QuickDraw provides a similar set of routines for drawing rounded rectangles, which are defined by their rectangles and the widths and heights of the ovals forming their corners. See "Drawing Rounded Rectangles" beginning on page 3-63 for detailed information about these routines.

Drawing Ovals, Arcs, and Wedges

An oval is a circular or elliptical shape defined by the bounding rectangle that encloses it. After specifying the bounding rectangle for an oval, you use the `FrameOval` procedure to draw its outline, or the `PaintOval` or `FillOval` procedure to draw its outline and its interior with a pattern. Figure 3-13 illustrates two ovals drawn with the `FrameOval` procedure.

Figure 3-13 Drawing ovals



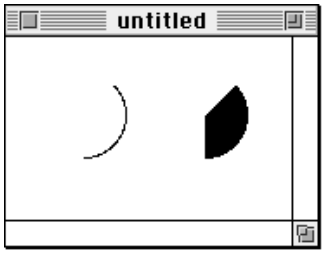
Listing 3-6 shows the code that produces the ovals in Figure 3-13. The bounding rectangles for the ovals are created with the `SetRect` procedure. The resulting rectangles are then passed to the `FrameOval` procedure.

Listing 3-6 Using the `FrameOval` procedure to draw ovals

```
PROCEDURE MyDrawOvals;
   VAR
      firstRect, secondRect: Rect;
BEGIN
   PenSize(2,2);
   SetRect(firstRect,20,20,70,70);   {create a bounding rectangle}
   FrameOval(firstRect);             {draw the oval}
   SetRect(secondRect,90,20,140,70); {create a bounding rectangle}
   FrameOval(secondRect);            {draw the oval}
   PenNormal;
END;
```

An arc is defined as a portion of an oval's circumference bounded by a pair of radii. A wedge is a pie-shaped segment bounded by a pair of radii, and it extends from the center of the oval to its circumference. You use the `FrameArc` procedure to draw an arc (as shown on the left side of Figure 3-14), and you use the `PaintArc` or `FillArc` procedure to draw a wedge (as shown on the right side of Figure 3-14).

Figure 3-14 Drawing an arc and a wedge



Listing 3-7 shows the code that produces the images in Figure 3-14. The `FrameArc`, `PaintArc`, and `FillArc` procedures take three parameters: a rectangle that defines an oval's boundaries, an angle indicating the start of the arc, and an angle indicating the arc's extent. For the angle parameters, 0° indicates a vertical line straight up from the center of the oval. Positive values indicate angles in the clockwise direction from this vertical line, and negative values indicate angles in the counterclockwise direction. The arc and the wedge in Figure 3-14 both begin at 45° and extend to 135°.

Listing 3-7 Using the `FrameArc` and `PaintArc` procedures

```
PROCEDURE MyDrawArcAndPaintWedge;
   VAR
      firstRect, secondRect: Rect;
BEGIN
   SetRect(firstRect,20,20,70,70);   {create a bounding rectangle}
   FrameArc(firstRect,45,135);       {draw an arc}
   SetRect(secondRect,90,20,140,70); {create a bounding rectangle}
   PaintArc(secondRect,45,135);      {draw a wedge}
END;
```

You can also fill, erase, and invert wedges by using, respectively, the `FillArc`, `EraseArc`, and `InvertArc` procedures.

Drawing Regions and Polygons

Before drawing regions and polygons, you must call several routines to create them. To create a region or polygon, you first call an open routine, which tells QuickDraw to collect subsequent routines to construct the shape. You use a close procedure when you are finished constructing the region or polygon. You can then frame the shape, fill it, paint it, erase it, and invert it.

To begin defining a region, you must use the `NewRgn` function to allocate space for it, and then call the `OpenRgn` procedure. You can then use any QuickDraw routine to construct the outline of the region. The outline can be any set of lines and shapes (including other regions) forming one or more closed loops. When you are finished constructing the region, use the `CloseRgn` procedure.

▲ **WARNING**
Ensure that the memory for a region is valid before calling routines to manipulate that region; if there isn't sufficient memory, the system may crash. Regions are limited to 32 KB in size in basic QuickDraw and 64 KB in Color QuickDraw. Before defining a region, you can use the Memory Manager function `MaxMem` to determine whether the memory for the region is valid. You can determine the current size of an existing region by calling the Memory Manager function `GetHandleSize`. (Both `MaxMem` and `GetHandleSize` are described in *Inside Macintosh: Memory*.) When you record drawing operations in an open region, the resulting region description may overflow the 32 KB or 64 KB limit. Should this happen in Color QuickDraw, the `QDError` function (described in the chapter "Color QuickDraw" in this book) returns the result code `regionTooBigError`. ▲

To draw the region, use the `FrameRgn`, `PaintRgn`, or `FillRgn` procedure. To draw the region with the background pattern of the graphics port, use the `EraseRgn` procedure; to invert the pixels in the region, use the `InvertRgn` procedure. When you no longer need the region, use the `DisposeRgn` procedure to release the memory used by the region.

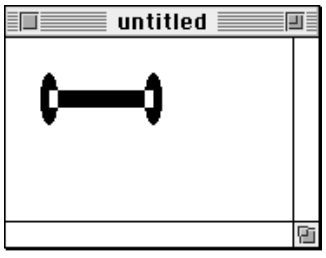
Listing 3-8 illustrates how to create and open a region, define a shape, close the region, fill it with the all-black pattern, and dispose of the region.

Listing 3-8 Creating and drawing a region

```
PROCEDURE MyDrawDumbbell;
VAR
   grow:       LongInt;
   dumbbell:   RgnHandle;
   tempRect:   Rect;
BEGIN
   IF MaxMem(grow) > kMinReserve THEN
   BEGIN
      dumbbell := NewRgn;           {create a new region}
      OpenRgn;                      {begin drawing instructions}
      SetRect(tempRect,20,20,30,50);
      FrameOval(tempRect);          {form the left "weight"}
      SetRect(tempRect,25,30,85,40);
      FrameRect(tempRect);          {form the bar}
      SetRect(tempRect,80,20,90,50);
      FrameOval(tempRect);          {form the right "weight"}
      CloseRgn(dumbbell);           {stop collecting}
      FillRgn(dumbbell,black);      {draw the shape onscreen}
      IF QDError <> noErr THEN
         ; {likely error is that there is insufficient memory}
      DisposeRgn(dumbbell)             {dispose of the region}
   END;
END;
```

Figure 3-15 shows the shape created by Listing 3-8.

Figure 3-15 A shape created by a region



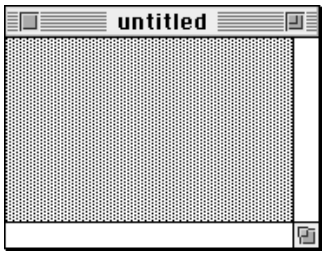
To assist you with scrolling, you can use QuickDraw routines to define a clipping region that excludes the scroll bars of the content region of a window. You can then scroll that area so that the region being updated does not draw into the scroll bars. Listing 3-9 illustrates how to create such a clipping region and, for illustrative purposes, how to fill it with a pattern. (The chapter "Basic QuickDraw" illustrates how to scroll the pixels in a rectangle such as the one created with the `ClipRect` procedure in Listing 3-9.)

Listing 3-9 Creating a clipping region and filling it with a pattern

```
FUNCTION MyFillClipRegion: RgnHandle;
VAR
   grow:          LongInt;
   newClip:       Rect;
   oldClipRegion: RgnHandle;
   newClipRegion: RgnHandle;
   myWindow:      WindowPtr;
BEGIN
   IF MaxMem(grow) > kMinReserve THEN
   BEGIN
      oldClipRegion := NewRgn;   {allocate old clipping region}
      myWindow := FrontWindow;   {get the front window}
      SetPort(myWindow);         {make the front window the current }
                                 { graphics port}
      GetClip(oldClipRegion);    {save the old clipping region}
      newClip := myWindow^.portRect;   {create a new rectangle}
      newClip.right := newClip.right - 15;   {exclude scroll bar}
      newClip.bottom := newClip.bottom - 15; {exclude scroll bar}
      ClipRect(newClip); {make the new rectangle the clipping region}
      newClipRegion := NewRgn;   {allocate new clipping region}
      RectRgn(newClipRegion, newClip);
      FillRgn(newClipRegion, ltGray);  {paint clipping region gray}
      SetClip(oldClipRegion);    {restore previous clipping region}
      IF QDError <> noErr THEN
         ; {likely error is that there is insufficient memory}
      DisposeRgn(oldClipRegion); {dispose previous clipping region}
      MyFillClipRect := (newClipRegion);
   END;
END;
```

Figure 3-16 shows the results of using the code in Listing 3-9.

Figure 3-16 Filling a clipping region



To create a polygon, you first call the `OpenPoly` function and then some number of `LineTo` procedures to draw lines from the first vertex of the polygon to the second, from the second to the third, and so on, until you've drawn a line to the last vertex. You then use the `ClosePoly` procedure, which completes the figure by drawing a connecting line from the last vertex back to the first. After defining a polygon in this way, you can display it with the `FramePoly`, `PaintPoly`, `FillPoly`, `ErasePoly`, and `InvertPoly` procedures. When you are finished using the polygon, use the `KillPoly` procedure to release its memory.

▲ **WARNING**
Do not create a height or width for the polygon greater than 32,767 pixels, or `PaintPoly` will crash. ▲

Listing 3-10 illustrates how to create a triangular polygon and fill it with a gray pattern.

Listing 3-10 Creating a triangular polygon

```
PROCEDURE MyDrawTriangle;
VAR
   triPoly: PolyHandle;
BEGIN
   triPoly := OpenPoly; {save handle and begin collecting lines}
   MoveTo(300,100);     {move to first point}
   LineTo(400,200);     {form the triangle's sides}
   LineTo(200,200);
   ClosePoly;           {stop collecting lines}
   FillPoly(triPoly,gray); {fill the polygon with gray}
   IF QDError <> noErr THEN
      ; {likely error is that there is insufficient memory}
   KillPoly(triPoly);   {dispose of its memory}
END;
```

Performing Calculations and Other Manipulations of Shapes

QuickDraw provides a multitude of routines for manipulating rectangles and regions. You can use the routines that manipulate rectangles to manipulate any shape based on a rectangle—namely, rounded rectangles, ovals, arcs, and wedges. For example, you could define a rectangle to bound an oval and then frame the oval. You could then use the `OffsetRect` procedure to move the oval's bounding rectangle downward. Using the offset bounding rectangle, you could frame a second, connected oval to form a figure eight with the first oval. You could use that shape to help define a region. You could create a second region, and then use the `UnionRgn` procedure to create a region from the union of the two.

The routines for performing calculations and other manipulations of rectangles are summarized in Table 3-3 and are described in detail in "Creating and Managing Rectangles" beginning on page 3-52.

Table 3-3 QuickDraw routines for calculating and manipulating rectangles

Routine	Description
`EmptyRect`	Determines whether a rectangle is an empty rectangle
`EqualRect`	Determines whether two rectangles are equal
`InsetRect`	Shrinks or expands a rectangle
`OffsetRect`	Moves a rectangle
`PtInRect`	Determines whether a pixel is enclosed in a rectangle
`PtToAngle`	Calculates the angle from the middle of a rectangle to a point
`Pt2Rect`	Determines the smallest rectangle that encloses two points
`SectRect`	Determines whether two rectangles intersect
`UnionRect`	Calculates the smallest rectangle that encloses two rectangles

The routines for performing calculations and other manipulations of regions are summarized in Table 3-4 and are described in detail in "Creating and Managing Regions" beginning on page 3-85.

Table 3-4 QuickDraw routines for calculating and manipulating regions

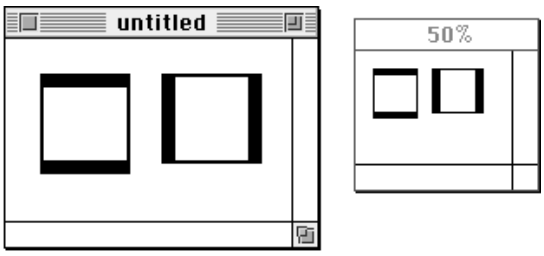
Routine	Description
`CopyRgn`	Makes a copy of a region
`DiffRgn`	Subtracts one region from another
`EmptyRgn`	Determines whether a region is empty
`EqualRgn`	Determines whether two regions have identical sizes, shapes, and locations
`InsetRgn`	Shrinks or expands a region
`OffsetRgn`	Moves a region
`PtInRgn`	Determines whether a pixel is within a region
`RectInRgn`	Determines whether a rectangle intersects a region
`RectRgn`	Changes a region to a rectangle
`SectRgn`	Calculates the intersection of two regions
`SetEmptyRgn`	Sets a region to empty
`SetRectRgn`	Changes a region to a rectangle
`UnionRgn`	Calculates the union of two regions
`XorRgn`	Calculates the difference between the union and the intersection of two regions

Note that while you can use the `OffSetPoly` procedure to move a polygon, QuickDraw provides no other routines for calculating or manipulating polygons.

Copying Bits Between Graphics Ports

You can use the `CopyBits` procedure to copy a bit image from one graphics port to another. Along with the `CopyMask` procedure and the Color QuickDraw procedure `CopyDeepMask`, `CopyBits` is integral to QuickDraw's image-processing capabilities. You can use `CopyBits` to move offscreen graphics images into an onscreen window, to blend colors for the image in a pixel map, and to shrink and expand images. For example, Figure 3-17 illustrates how `CopyBits` can be used to scale the image in one window to a smaller image in another window.

Figure 3-17 Shrinking images between graphics ports



Listing 3-11 shows the code that produces the scaled image in Figure 3-17.

Listing 3-11 Using the `CopyBits` procedure to copy between two windows

```
PROCEDURE MyShrinkImages;
   VAR
      myWindow:               WindowPtr;
      sourceRect, destRect:   rect;
      halfHeight, halfWidth:  Integer;
BEGIN
   myWindow := FrontWindow;
   sourceRect.top := myWindow^.portRect.top; {create source rectangle}
   sourceRect.left := myWindow^.portRect.left;
   sourceRect.bottom := myWindow^.portRect.bottom - 15; {exclude scroll bar}
   sourceRect.right := myWindow^.portRect.right - 15;  {exclude scroll bar}

   destRect.top := gShrinkWindow^.portRect.top; {create destination rect}
   destRect.left := gShrinkWindow^.portRect.left;
   halfHeight :=               {make destination half as tall as the source}
            Integer((sourceRect.bottom - sourceRect.top)) DIV 2;
   destRect.bottom := destRect.top + halfHeight;
   halfWidth :=                {make destination half as wide as the source}
            Integer((sourceRect.right - sourceRect.left)) DIV 2;
   destRect.right := destRect.left + halfWidth;

   GetPort(myWindow);      {save the graphics port for the active window}
   SetPort(gShrinkWindow); {make the target window the current }
                           { graphics port for drawing purposes}
```

```
   CopyBits(myWindow^.portBits,
            gShrinkWindow^.portBits,
            sourceRect,
            destRect,
            srcCopy+ditherCopy, NIL);
   IF QDError <> noErr THEN
      ; {likely error is that there is insufficient memory}
   SetPort(myWindow);      {restore active window as current graphics port}
END;
```

When copying between basic graphics ports, you specify a source bitmap and a destination bitmap to `CopyBits`. Remember that the bitmap is stored in the `portBits` field of a `GrafPort` record. By dereferencing the desired window record when it calls `CopyBits`, Listing 3-11 uses the bitmap for the front window, "untitled" in Figure 3-17, as the source bitmap. Listing 3-11 uses the bitmap for the window titled "50%" as the destination bitmap.

When copying images between color graphics ports, as explained in the chapter "Color QuickDraw," you must coerce each `CGrafPort` record to a `GrafPort` record, dereference the `portBits` fields of each, and then pass these "bitmaps" to `CopyBits`.

Note
If there is insufficient memory to complete a `CopyBits` operation in Color QuickDraw, the `QDError` function (described in the chapter "Color QuickDraw" in this book) returns the result code –143. ◆

You can specify differently sized source and destination rectangles, and `CopyBits` scales the source image to fit the destination. Listing 3-11 uses the area of the port rectangle excluding the scroll bars as the source rectangle. To scale the image in the front window, Listing 3-11 creates a destination rectangle that is half as high and half as wide as the source rectangle.

The manner by which `CopyBits` transfers the bits between bitmaps depends on the source mode that you specify. In Listing 3-11, the `srcCopy` mode is used to copy bits from the source directly into the destination. Source modes are described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8.

Note
To scale shapes and regions within the same graphics port, you can use the routines described in "Scaling and Mapping Points, Rectangles, Polygons, and Regions" beginning on page 3-104. ◆

To gracefully display complex images that your application creates, your application should use the drawing routines described in this chapter to construct such images in offscreen graphics worlds. Your application can then use the `CopyBits` procedure to transfer these images to onscreen graphics ports. This technique prevents the choppiness that can occur when you build complex images onscreen, and is described in the chapter "Offscreen Graphics Worlds," which also offers an example of using a mask to copy color pixels from an offscreen graphics world.

To copy only certain bits from a bitmap, you can use the `CopyMask` procedure, which is a specialized variant of `CopyBits`. The `CopyMask` procedure, which is described on page 3-119, transfers bits only where the corresponding bits of another bit image, which serves as a mask, are set to 1 (that is, black). The `CopyMask` procedure does not allow scaling or resizing. However, the `CopyDeepMask` procedure, which is described on page 3-120, does allow scaling and resizing; in effect it combines the capabilities of the `CopyBits` and `CopyMask` procedures.

Customizing QuickDraw's Low-Level Routines

For each shape that QuickDraw knows how to draw, there are procedures that perform these basic graphics operations on the shape: frame, paint, erase, invert, and fill. Those procedures in turn call a low-level drawing routine for the shape. For example, the `FrameOval`, `PaintOval`, `EraseOval`, `InvertOval`, and `FillOval` procedures all call the low-level procedure `StdOval`, which draws the oval. For each type of object QuickDraw can draw, including text and lines, there's a pointer to such a low-level routine. By changing these pointers, you can install your own routines, and either completely override the standard ones or call them after your routines have modified their parameters as necessary.

Other low-level routines that you can install in this way include

- The procedure (called by `CopyBits`) that performs bit and pixel transfer.
- The function that measures the width of text and is called by the QuickDraw text routines `CharWidth`, `StringWidth`, and `TextWidth`. (These QuickDraw text routines are described in the chapter "QuickDraw Text" in *Inside Macintosh: Text*.)
- The procedure that processes picture comments. The standard procedure ignores picture comments. (Picture comments are described in Appendix B of this book.)
- The procedure that saves drawing commands as the definition of a picture, and the procedure that retrieves them. These enable your application to draw on remote devices, print to the disk, get picture input from the disk, and support large pictures.

All of the low-level QuickDraw routines that your application can replace or call after performing its own operations are described in "Customizing QuickDraw Operations" beginning on page 3-129.

The `grafProcs` field of a graphics port determines which low-level routines are called. If that field contains the value of `NIL`, the standard routines are called, so that all operations in that graphics port are done in the standard ways described in this chapter. You can set the `grafProcs` field to point to a record of pointers to your own routines. This record of pointers is defined by a data structure of type `QDProcs`, which is described on page 3-39.

To assist you in setting up a record, QuickDraw provides the `SetStdProcs` procedure, which is described on page 3-130. You can use the `SetStdProcs` procedure to get a `QDProcs` record with fields that point to the standard routines. You can reset the ones with which you are concerned. You can replace these low-level routines with your own, and then point to your modified `QDProcs` record in the `grafProcs` field of a `GrafPort` record to change basic QuickDraw's standard low-level behavior.

IMPORTANT
When modifying the low-level routines for a color graphics port, you must always use the `SetStdCProcs` procedure instead of `SetStdProcs`. ▲

The chapter "Pictures" in this book provides sample code and explanations for changing the standard low-level routines for reading and writing pictures.

QuickDraw Drawing Reference

This section describes the data structures, routines, and resources that QuickDraw provides to assist you in drawing lines and shapes onscreen.

"Data Structures" shows the Pascal data structures for the `Polygon`, `PenState`, `QDProcs`, and `Pattern` data types.

"Routines" describes QuickDraw routines for drawing lines, rectangles, rounded rectangles, ovals, arcs, wedges, polygons, and regions. "Routines" also describes routines for calculating, scaling, mapping, copying bits between, and otherwise manipulating these graphic entities.

"Resources" describes the pattern and pattern list resources, which your application can create to define its own bit patterns for drawing lines and shapes.

Data Structures

This section describes the `Polygon` record, the `PenState` record, the `QDProcs` record, and the `Pattern` record. Although this chapter describes routines for creating and manipulating rectangles and regions, the data structures you can use to define these entities are described in the chapter "Basic QuickDraw."

Your application typically does not create `Polygon` or `Pattern` records. Instead, you use the `OpenPoly` function (described on page 3-78) to create a polygon, and QuickDraw creates the necessary record. Although you can create a `Pattern` record in your program code, it is usually easier to create patterns using the pattern or pattern list resources, which are described beginning on page 3-140.

You need to use the `QDProcs` record only if you customize one or more of QuickDraw's low-level drawing routines. You can use the `SetStdProcs` procedure, described on page 3-130, to create a `QDProcs` record.

You can use a `PenState` record to save the location, size, pattern, and pattern mode of a graphics pen. The `GetPenState` procedure, described on page 3-43, automatically creates a pen state record. You can use the `SetPenState` procedure, described on page 3-43, to restore the values stored in a `PenState` record to the current graphics pen.

Polygon

After you use the `OpenPoly` function to create a polygon, QuickDraw begins collecting the line-drawing information you provide into a `Polygon` record, which is a data structure of type `Polygon`. The `OpenPoly` function returns a handle to the newly allocated `Polygon` record. Thereafter, your application normally refers to your new polygon by this handle, because QuickDraw routines such as `FramePoly` and `PaintPoly` expect a handle to a `Polygon` record as their first parameter.

A polygon is defined by a sequence of connected lines. A `Polygon` record consists of two fixed-length fields followed by a variable-length array of points: the starting point followed by each successive point to which a line is drawn.

```
Polygon =
   RECORD
      polySize:   Integer; {size in bytes}
      polyBBox:   Rect;    {bounding rectangle}
      polyPoints: ARRAY[0..0] OF Point;      {vertices of polygon}
   END;
```

Field descriptions

`polySize`	The size in bytes of this record.
`polyBBox`	The rectangle that bounds the polygon.
`polyPoints`	An array of points: the starting point followed by each successive point to which a line is drawn.

PenState

The `GetPenState` procedure (described on page 3-43) saves the location, size, pattern, and pattern mode of the graphics pen for the current graphics port in a `PenState` record, which is a data structure of type `PenState`. After changing the graphics pen as necessary, you can later restore these pen states with the `SetPenState` procedure (described on page 3-43).

Here is how a `PenState` record is defined:

```
TYPE PenState =
   RECORD
      pnLoc:   Point;      {pen location}
      pnSize:  Point;      {pen size}
      pnMode:  Integer;    {pen's pattern mode}
      pnPat:   Pattern;    {pen pattern}
   END;
```

Field descriptions

`pnLoc` For the current graphics port at the time the `GetPenState` procedure was called, the value of that graphics port's `pnLoc` field. This value is the point where QuickDraw begins drawing next. 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.

`pnSize` For the current graphics port at the time the `GetPenState` procedure was called, the value of that graphics port's `pnSize` field. The graphics pen is rectangular in shape, and its width and height are specified by the values in the `pnSize` field. The default size is a 1-by-1 bit square; the width and height 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.

`pnMode` The pattern mode—that is, for the current graphics port at the time the `GetPenState` procedure was called, the value of that graphics port's `pnMode` field. This value determines how the pen pattern is to affect what's already in the bit image when lines or shapes are drawn. When the graphics pen draws, QuickDraw first determines what bits in the bit image are affected, finds their corresponding bits in the pattern, and then transfers the bits from the pattern into the image according to this mode, which specifies one of eight Boolean transfer operations. The resulting bit is stored into its proper place in the bit image. The various pattern modes are described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8.

`pnPat` For the current graphics port at the time the `GetPenState` procedure was called, the pen pattern for that graphics port. This pattern determines how the bits under the graphics pen are affected when lines or shapes are drawn.

QDProcs

You need to use the `QDProcs` record, which is a data structure of type `QDProcs`, only if you customize one or more of QuickDraw's low-level drawing routines. You can use the `SetStdProcs` procedure, described on page 3-130, to create a `QDProcs` record.

The `QDProcs` record contains pointers to low-level drawing routines. QuickDraw's standard low-level drawing routines are described in "Customizing QuickDraw Operations" beginning on page 3-129. You can change the fields of this record to point to routines of your own devising.

```
TYPE QDProcsPtr = ^QDProcs;
QDProcs =
   RECORD
      textProc:   Ptr;  {text drawing}
      lineProc:   Ptr;  {line drawing}
      rectProc:   Ptr;  {rectangle drawing}
      rRectProc:  Ptr;  {roundRect drawing}
      ovalProc:   Ptr;  {oval drawing}
      arcProc:    Ptr;  {arc/wedge drawing}
      polyProc:   Ptr;  {polygon drawing}
      rgnProc:    Ptr;  {region drawing}
      bitsProc:   Ptr;  {bit transfer}
      commentProc:Ptr;  {picture comment processing}
      txMeasProc: Ptr;  {text width measurement}
      getPicProc: Ptr;  {picture retrieval}
      putPicProc: Ptr;  {picture saving}
    END;
```

Field descriptions

`textProc` A pointer to the low-level routine that draws text. The standard QuickDraw routine is the `StdText` procedure.

`lineProc` A pointer to the low-level routine that draws lines. The standard QuickDraw routine is the `StdLine` procedure.

`rectProc` A pointer to the low-level routine that draws rectangles. The standard QuickDraw routine is the `StdRect` procedure.

`rRectProc` A pointer to the low-level routine that draws rounded rectangles. The standard QuickDraw routine is the `StdRRect` procedure.

`ovalProc` A pointer to the low-level routine that draws ovals. The standard QuickDraw routine is the `StdOval` procedure.

`arcProc` A pointer to the low-level routine that draws arcs. The standard QuickDraw routine is the `StdArc` procedure.

`polyProc` A pointer to the low-level routine that draws polygons. The standard QuickDraw routine is the `StdPoly` procedure.

`rgnProc`	A pointer to the low-level routine that draws regions. The standard QuickDraw routine is the `StdRgn` procedure.
`bitsProc`	A pointer to the low-level routine that copies bitmaps. The standard QuickDraw routine is the `StdBits` procedure.
`commentProc`	A pointer to the low-level routine for processing a picture comment. The standard QuickDraw routine is the `StdComment` procedure.
`txMeasProc`	A pointer to the low-level routine for measuring text width. The standard QuickDraw routine is the `StdTxtMeas` function.
`getPicProc`	A pointer to the low-level routine for retrieving information from the definition of a picture. The standard QuickDraw routine is the `StdGetPic` procedure.
`putPicProc`	A pointer to the low-level routine for saving information as the definition of a picture. The standard QuickDraw routine is the `StdPutPic` procedure.

You can use the `SetStdProcs` procedure to set all the fields of the `QDProcs` record to point to QuickDraw's standard routines, and then reset the ones for which you have your own routines.

Pattern

Your application typically does not create `Pattern` records, which are data structures of type `Pattern`. Although you can create `Pattern` records in your program code, it is usually easier to create bit patterns using the pattern or pattern list resource, described beginning on page 3-140.

A bit pattern is a 64-bit image, organized as an 8-by-8 bit square, that defines a repeating design or tone. When a pattern is drawn, it's aligned so that adjacent areas of the same pattern in the same graphics port form a continuous, coordinated pattern. QuickDraw provides predefined patterns in global variables named `white`, `black`, `gray`, `ltGray`, and `dkGray`. A `Pattern` record is defined as follows:

```
TYPE  PatPtr      = ^Pattern;
      PatHandle   = ^PatPtr;
      Pattern     = PACKED ARRAY[0..7] OF 0..255;
```

The row width of a pattern is 1 byte.

Routines

This section describes QuickDraw's routines for drawing lines, rectangles, rounded rectangles, ovals, arcs, wedges, polygons, and regions. This section also describes routines for calculating, scaling, mapping, copying bits between, and otherwise manipulating these graphic entities.

These routines use your current graphics port as their drawing environment. You can use these routines to draw into basic graphics ports (described in the chapter "Basic QuickDraw") and color graphics ports (described in the chapter "Color QuickDraw").

See the chapter "Pictures" for descriptions of routines that create and draw pictures. See the chapter "Cursor Utilities" for information about drawing cursors. See the chapter "QuickDraw Text" in *Inside Macintosh: Text* for descriptions of QuickDraw's routines for drawing and manipulating text.

Managing the Graphics Pen

Every graphics port contains one, and only one, graphics pen with which to perform drawing operations. You use this metaphorical pen to draw lines, shapes, and text.

You can use the `HidePen` and `ShowPen` procedures to change the pen's visibility, the `PenSize` procedure to change its shape, the `PenPat` procedure to change its pattern, and the `PenMode` procedure to change its pattern mode. To determine the size, location, pattern, and pattern mode of the graphics pen, you can use the `GetPenState` procedure. If you need to temporarily change these characteristics, you can use the `SetPenState` procedure to restore the graphics pen to a state previously captured by `GetPenState`.

Upon the creation of a graphics port, QuickDraw assigns these initial values to the graphics pen: a size of (1,1), a pattern of all-black pixels, and the `patCopy` pattern mode. After changing any of these values, you can use the `PenNormal` procedure to return these initial values to the graphics pen.

These pen-manipulation routines use the local coordinate system of the current graphics port. Remember that each graphics port has its own pen, the state of which is stored in several fields of its `GrafPort` or `CGrafPort` record. If you draw in one graphics port, change to another, and return to the first, the pen for the first graphics port has the same state as when you left it. (The basic graphics port is described in the chapter "Basic QuickDraw," and the color graphics port is described in the chapter "Color QuickDraw.")

HidePen

To give the graphics pen invisible ink (which means that pen drawing doesn't show on the screen), use the `HidePen` procedure.

```
PROCEDURE HidePen;
```

DESCRIPTION

The `HidePen` procedure decrements the `pnVis` field of the current graphics port. The `pnVis` field is initialized to 0 by the `OpenPort` procedure. Whenever `pnVis` is negative, the pen doesn't draw on the screen. The `pnVis` field keeps track of the number of times the pen has been hidden to compensate for nested calls to the `HidePen` and `ShowPen` routines.

Every call to `HidePen` should be balanced by a subsequent call to `ShowPen`, which is described next.

The `HidePen` procedure is called by the `OpenRgn`, `OpenPicture`, and `OpenPoly` routines so that you can create regions, pictures, and polygons without drawing on the screen.

ShowPen

To change the ink of a graphics pen from invisible (which means that pen drawing doesn't show on the screen) to visible (so that pen drawing does appear on the screen), you can use the `ShowPen` procedure.

```
PROCEDURE ShowPen;
```

DESCRIPTION

The `ShowPen` procedure increments the `pnVis` field of the current graphics port. For 0 or positive values, pen drawing shows on the screen.

For example, if you have used the `HidePen` procedure to decrement the `pnVis` field from 0 to –1, you can use the `ShowPen` procedure to make its value 0 so that QuickDraw resumes drawing on the screen. Subsequent calls to `ShowPen` increment `pnVis` beyond 0, so every call to `ShowPen` should be balanced by a call to `HidePen`.

`ShowPen` is called by the procedures `CloseRgn` (described on page 3-89), `ClosePoly` (described on page 3-79), and `ClosePicture` (described in the chapter "Pictures" in this book).

GetPen

To determine the location of the graphics pen, use the `GetPen` procedure.

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

`pt` The graphics pen's current position in the current graphics port.

DESCRIPTION

In the `pt` parameter, the `GetPen` procedure returns the current pen position. The point returned is in the local coordinates of the current graphics port.

GetPenState

To determine the graphics pen's location, size, pattern, and pattern mode, you can use the `GetPenState` procedure.

```
PROCEDURE GetPenState (VAR pnState: PenState);
```

`pnState` A `PenState` record for holding information about the graphics pen.

DESCRIPTION

The `GetPenState` procedure saves the location, size, pattern, and pattern mode of the graphics pen for the current graphics port in a `PenState` record, which the `GetPenState` procedure returns in the `pnState` parameter.

After changing the graphics pen as necessary, you can later restore these pen states with the `SetPenState` procedure (described next).

SEE ALSO

The `PenState` record is described on page 3-37.

SetPenState

To restore the state of the graphics pen that was saved with the `GetPenState` procedure, use the `SetPenState` procedure.

```
PROCEDURE SetPenState (pnState: PenState);
```

`pnState` A `PenState` record previously created with the `GetPenState` procedure.

DESCRIPTION

The `SetPenState` procedure sets the graphics pen's location, size, pattern, and pattern mode in the current graphics port to the values stored in the `PenState` record that you specify in the `pnState` parameter.

SEE ALSO

The `PenState` record is described on page 3-37.

PenSize

To set the dimensions of the graphics pen in the current graphics port, use the `PenSize` procedure.

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

`width` The pen width, as an integer from 0 to 32,767. If you set the value to 0, the pen does not draw. Values less than 0 are undefined.

`height` The pen height, as an integer from 0 to 32,767. If you set the value to 0, the pen does not draw. Values less than 0 are undefined.

DESCRIPTION

The `PenSize` procedure sets the width that you specify in the `width` parameter and the height that you specify in the `height` parameter for the graphics pen in the current graphics port. All subsequent calls to the `Line` and `LineTo` procedures and to the procedures that draw framed shapes in the current graphics port use the new pen dimensions.

You can get the current pen dimensions from the `pnSize` field of the current graphics port, where the width and height are stored as a `Point` record.

SEE ALSO

Listing 3-2 on page 3-20 illustrates how to use this procedure.

PenMode

To set the pattern mode of the graphics pen in the current graphics port, use the `PenMode` procedure.

```
PROCEDURE PenMode (mode: Integer);
```

`mode` The pattern mode. The following list shows the constants you can use—and the values they represent—for specifying the pattern mode.

```
CONST
   patCopy     = 8;  {where pattern pixel is black, apply }
                     { foreground color to destination pixel; }
                     { where pattern pixel is white, apply }
                     { background color to destination pixel}
   patOr       = 9;  {where pattern pixel is black, invert }
                     { destination pixel; where pattern }
                     { pixel is white, leave }
                     { destination pixel unaltered}
   patXor      = 10; {where pattern pixel is black, invert }
                     { destination pixel; where pattern }
                     { pixel is white, leave destination }
                     { pixel unaltered}
   patBic      = 11; {where pattern pixel is black, apply }
                     { background color to destination pixel; }
                     { where pattern pixel is white, leave }
                     { destination pixel unaltered}
   notPatCopy  = 12; {where pattern pixel is black, apply }
                     { background color to destination pixel; }
                     { where pattern pixel is white, apply }
                     { foreground color to destination pixel}
   notPatOr    = 13; {where pattern pixel is black, leave }
                     { destination pixel unaltered; where }
                     { pattern pixel is white, apply }
                     { foreground color to destination pixel}
   notPatXor   = 14; {where pattern pixel is black, }
                     { leave destination pixel unaltered; }
                     { where pattern pixel is white, }
                     { invert destination pixel}
   notPatBic   = 15; {where pattern pixel is black, }
                     { leave destination pixel unaltered; }
                     { where pattern pixel is white, apply }
                     { background color to destination pixel}
```

DESCRIPTION

Using the pattern mode you specify in the `mode` parameter, the `PenMode` procedure sets the manner in which the pattern of the graphics pen is transferred onto the bitmap (or pixel map) when you draw lines or shapes in the current graphics port. These actions are illustrated in Figure 3-4 on page 3-10.

If you specify a source mode (such as one used with the `CopyBits` procedure) instead of a pattern mode, no drawing is performed.

The current pattern mode is stored in the `pnMode` field of the current graphics port. The initial pattern mode value is `patCopy`, in which the pen pattern is copied directly to the bitmap.

To use highlighting, you can add this constant or its value to the source or pattern mode:

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

With highlighting, QuickDraw replaces the background color with the highlight color when your application draws or copies images between graphics ports. This has the visual effect of using a highlighting pen to select the object. (The global variable `HiliteRGB` is read from parameter RAM when the machine starts. Basic graphics ports use the color stored in the `HiliteRGB` global variable as the highlight color. Color graphics ports default to the `HiliteRGB` global variable, but can be overridden by the `HiliteColor` procedure, described in the chapter "Color QuickDraw.")

SPECIAL CONSIDERATIONS

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

The results of inverting a pixel are predictable only with direct pixels or 1-bit pixel maps. For indexed pixels, Color QuickDraw performs the inversion on the pixel indexes, which means the results depend entirely on the contents of the color table (which is described in the chapter "Color QuickDraw"). The eight colors used in basic QuickDraw are stored in a color table represented by the global variable `QDColors`. To display those eight basic QuickDraw colors on an indexed device, Color QuickDraw uses the Color Manager to obtain indexes to the colors in the CLUT that best map to the colors in the `QDColors` color table. Because the index, not the color value, is inverted, the results are unpredictable.

SEE ALSO

Pattern modes are discussed in detail in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8 of this chapter and in "Boolean Transfer Modes With Color Pixels" beginning on page 4-32 in the chapter "Color QuickDraw."

PenPat

To set the bit pattern to be used by the graphics pen in the current graphics port, use the `PenPat` procedure.

```
PROCEDURE PenPat (pat: Pattern);
```

`pat` A bit pattern, as defined by a `Pattern` record.

DESCRIPTION

The `PenPat` procedure sets the graphics pen to use the bit pattern defined in the `Pattern` record that you specify in the `pat` parameter. (The standard patterns `white`, `black`, `gray`, `ltGray`, and `dkGray` are predefined; the initial bit pattern for the pen is `black`.) This pattern is stored in the `pnPat` field of a `GrafPort` record. The QuickDraw painting procedures (such as `PaintRect`) also use the pen's pattern when drawing a shape.

The `PenPat` procedure also sets a bit pattern for the graphics pen in a color graphics port. The `PenPat` procedure creates a handle, of type `PixPatHandle`, for the bit pattern and stores this handle in the `pnPixPat` field of the `CGrafPort` record. This pattern always uses the graphics port's current foreground and background colors.

SPECIAL CONSIDERATIONS

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

SEE ALSO

The `Pattern` record is described on page 3-40. To define your own patterns, you typically create pattern or pattern list resources, which are described beginning on page 3-140.

The `CGrafPort` record is described in the chapter "Color QuickDraw." To set the graphics pen to use a multicolored pixel pattern in a color graphics port, use the `PenPixPat` procedure, which is described in the chapter "Color QuickDraw."

Listing 3-3 on page 3-21 illustrates how to use the `PenPat` procedure.

PenNormal

To set the size, pattern, and pattern mode of the graphics pen in the current graphics port to their initial values, use the `PenNormal` procedure.

```
PROCEDURE PenNormal;
```

DESCRIPTION

The `PenNormal` procedure restores the size, pattern, and pattern mode of the graphics pen in the current graphics port to their initial values: a size of 1 pixel by 1 pixel, a pattern mode of `patCopy`, and a pattern of `black`. The pen location does not change.

SPECIAL CONSIDERATIONS

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

Changing the Background Bit Pattern

Each graphics port has a background pattern that's used when an area is erased (such as by using the `EraseRect`, `EraseRoundRect`, `EraseArc`, `ErasePoly`, and `EraseRgn` procedures) and when pixels are scrolled out of an area (such as by using the `ScrollRect` procedure described in the chapter "Basic QuickDraw"). The background pattern is stored in the `bkPat` field of every basic graphics port. You can use the `BackPat` procedure to change the bit pattern used as the background color by the current graphics port (black and white or color).

BackPat

To change the bit pattern used as the background pattern by the current graphics port, use the `BackPat` procedure.

```
PROCEDURE BackPat (pat: Pattern);
```

`pat` A bit pattern, as defined by a `Pattern` record.

DESCRIPTION

The `BackPat` procedure sets the bit pattern defined in the `Pattern` record, which you specify in the `pat` parameter, to be the background pattern. (The standard bit patterns `white`, `black`, `gray`, `ltGray`, and `dkGray` are predefined; the initial background pattern for the graphics port is `white`.) This pattern is stored in the `bkPat` field of a `GrafPort` record.

The `BackPat` procedure also sets a bit pattern for the background color in a color graphics port. The `BackPat` procedure creates a handle, of type `PixPatHandle`, for the bit pattern and stores this handle in the `bkPixPat` field of the `CGrafPort` record. As in basic graphics ports, Color QuickDraw draws patterns in color graphics ports at the time of drawing, not at the time you use `PenPat` to set the pattern.

SPECIAL CONSIDERATIONS

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

SEE ALSO

The `Pattern` record is described on page 3-40. To define your own patterns, you typically create pattern or pattern list resources, which are described beginning on page 3-140.

The `CGrafPort` record is described in the chapter "Color QuickDraw." To use a multicolored background pattern in a color graphics port, use the `BackPixPat` procedure, which is described in the chapter "Color QuickDraw."

Drawing Lines

A line is defined by two points: the current location of the graphics pen and its destination. You specify where to begin drawing a line by using the `MoveTo` or `Move` procedure to place the graphics pen at some point in the window's local coordinate system, and then using the `LineTo` or `Line` procedure to draw a line from there to another point.

MoveTo

To move the graphics pen to a particular location in the current graphics port, use the `MoveTo` procedure.

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

`h`	The horizontal coordinate of the graphics pen's new position.
`v`	The vertical coordinate of the graphics pen's new position.

DESCRIPTION

The `MoveTo` procedure changes the graphics pen's current location to the new horizontal coordinate you specify in the `h` parameter and the new vertical coordinate you specify in the `v` parameter. Specify the new location in the local coordinates of the current graphics port. The `MoveTo` procedure performs no drawing.

SEE ALSO

Listing 3-1 on page 3-18 illustrates how to use this procedure.

Move

To move the graphics pen a particular distance, use the `Move` procedure.

```
PROCEDURE Move (dh,dv: Integer);
```

`dh`	The horizontal distance of the graphics pen's movement.
`dv`	The vertical distance of the graphics pen's movement.

DESCRIPTION

The `Move` procedure moves the graphics pen from its current location in the current graphics port a horizontal distance that you specify in the `dh` parameter and a vertical distance that you specify in the `dv` parameter. The `Move` procedure calls

```
MoveTo(h+dh,v+dv)
```

where `(h,v)` is the graphics pen's current location in local coordinates. The `Move` procedure performs no drawing.

SEE ALSO

Listing 3-1 on page 3-18 illustrates how to use this procedure.

LineTo

To draw a line from the graphics pen's current location to a new location, use the `LineTo` procedure.

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

`h`	The horizontal coordinate of the graphics pen's new location.
`v`	The vertical coordinate of the graphics pen's new location.

DESCRIPTION

The `LineTo` procedure draws a line from the graphics pen's current location in the current graphics port to the new location `(h,v)`, which you specify in the local coordinates of the current graphics port. If you are using `LineTo` to draw a region or polygon, its outline is infinitely thin and is not affected by the values of the `pnSize`, `pnMode`, or `pnPat` field of the graphics port.

SPECIAL CONSIDERATIONS

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

SEE ALSO

Listing 3-1 on page 3-18 illustrates how to use this procedure.

Line

To draw a line a specified distance from the graphics pen's current location in the current graphics port, use the `Line` procedure.

```
PROCEDURE Line (dh,dv: Integer);
```

`dh`	The horizontal distance of the graphics pen's movement.
`dv`	The vertical distance of the graphics pen's movement.

DESCRIPTION

Starting at the current location of the graphics pen, the `Line` procedure draws a line the horizontal distance that you specify in the `dh` parameter and the vertical distance that you specify in the `dv` parameter. The `Line` procedure calls

```
LineTo(h+dh,v+dv)
```

where `(h,v)` is the current location in local coordinates. The pen location becomes the coordinates of the end of the line after the line is drawn. If you are using `Line` to draw a region or polygon, its outline is infinitely thin and is not affected by the values of the `pnSize`, `pnMode`, and `pnPat` fields of the graphics port.

SPECIAL CONSIDERATIONS

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

SEE ALSO

Listing 3-1 on page 3-18 illustrates how to use this procedure.

Creating and Managing Rectangles

You can use a rectangle, which is defined by a `Rect` record, to specify locations and sizes for various graphics operations. (The `Rect` data type is described in the chapter "Basic QuickDraw.") You can use the `SetRect` procedure to create a rectangle, `OffsetRect` to move one, and `InsetRect` to shrink or expand one. You can determine whether two rectangles intersect with the `SectRect` procedure, whether a pixel is enclosed in a rectangle with the `PtInRect` procedure, whether two rectangles are equal with the `EqualRect` procedure, and whether a rectangle is an empty rectangle with the `EmptyRect` procedure. You can use the `UnionRect` procedure to calculate the smallest rectangle that encloses two other rectangles, `PtToAngle` to calculate the angle from the middle of a rectangle to a point, and `Pt2Rect` to determine the smallest rectangle that encloses two points.

If the points or rectangles supplied to these routines are defined in a graphics port other than your current graphics port, you must convert them to the local coordinate system of your current graphics port. You can accomplish this by using the `SetPort` procedure to change to the graphics port containing the points or rectangles, using the `LocalGlobal` procedure to convert their locations to global coordinates, using `SetPort` to return to your starting graphics port, and then using the `GlobalToLocal` procedure to convert the locations of points or rectangles to the local coordinates of your current graphics port. These procedures are described in the chapter "Basic QuickDraw."

SetRect

To assign coordinates to a rectangle, you can use the `SetRect` procedure.

```
PROCEDURE SetRect (VAR r: Rect; left,top,right,bottom: Integer);
```

`r`	The rectangle to set.
`left`	The horizontal coordinate of the new upper-left corner of the rectangle.
`top`	The vertical coordinate of the new upper-left corner of the rectangle.
`right`	The horizontal coordinate of the new lower-right corner of the rectangle.
`bottom`	The vertical coordinate of the new lower-right corner of the rectangle.

DESCRIPTION

The `SetRect` procedure assigns the coordinates you specify in the `left`, `top`, `right`, and `bottom` parameters to the rectangle that you specify in the `r` parameter. This procedure is provided to help you shorten your program text. If you want a more readable text, at the expense of source text length, you can instead assign integers (or points) directly into the fields of a `Rect` record.

SEE ALSO

Listing 3-4 on page 3-23 illustrates how to use this procedure. The data structure of type `Rect` is described in the chapter "Basic QuickDraw."

OffsetRect

To move a rectangle, use the `OffsetRect` procedure.

```
PROCEDURE OffsetRect (VAR r: Rect; dh,dv: Integer);
```

`r`	The rectangle to move.
`dh`	The horizontal distance to move the rectangle.
`dv`	The vertical distance to move the rectangle.

DESCRIPTION

The `OffsetRect` procedure moves the rectangle that you specify in the `r` parameter by adding the value you specify in the `dh` parameter to each of its horizontal coordinates and the value you specify in the `dv` parameter to each of its vertical coordinates. If the `dh` and `dv` parameters are positive, the movement is to the right and down; if either is negative, the corresponding movement is in the opposite direction. The rectangle retains its shape and size; it's merely moved on the coordinate plane. The movement doesn't affect the screen unless you subsequently call a routine to draw within the rectangle.

InsetRect

To shrink or expand a rectangle, use the `InsetRect` procedure.

```
PROCEDURE InsetRect (VAR r: Rect; dh,dv: Integer);
```

`r`	The rectangle to alter.
`dh`	The horizontal distance to move the left and right sides in toward or outward from the center of the rectangle.
`dv`	The vertical distance to move the top and bottom sides in toward or outward from the center of the rectangle.

DESCRIPTION

The `InsetRect` procedure shrinks or expands the rectangle that you specify in the `r` parameter: the left and right sides are moved in by the amount you specify in the `dh` parameter; the top and bottom are moved toward the center by the amount you specify in the `dv` parameter. If the value you pass in `dh` or `dv` is negative, the appropriate pair of sides is moved outward instead of inward. The effect is to alter the size by `2*dh` horizontally and `2*dv` vertically, with the rectangle remaining centered in the same place on the coordinate plane.

If the resulting width or height becomes less than 1, the rectangle is set to the empty rectangle (0,0,0,0).

SectRect

To determine whether two rectangles intersect, you can use the `SectRect` function.

```
FUNCTION SectRect (src1,src2: Rect; VAR dstRect: Rect): Boolean;
```

`src1`	The first of two rectangles to test for intersection.
`src2`	The second of two rectangles to test for intersection.
`dstRect`	The rectangle marking the intersection of the first two rectangles.

DESCRIPTION

The `SectRect` function calculates the rectangle that delineates the intersection of the two rectangles you specify in the `src1` and `src2` parameters. The `SectRect` function returns the area of intersection in the `dstRect` parameter and a function result of `TRUE` if they intersect or `FALSE` if they don't. Rectangles that touch at a line or a point are not considered intersecting, because their intersection rectangle (actually, in this case, an intersection line or point) doesn't enclose any pixels in the bit image.

If the rectangles don't intersect, the destination rectangle is set to (0,0,0,0). The `SectRect` procedure works correctly even if one of the source rectangles is also the destination.

UnionRect

To calculate the smallest rectangle that encloses two rectangles, use the `UnionRect` procedure.

```
PROCEDURE UnionRect (src1,src2: Rect; VAR dstRect: Rect);
```

`src1`	The first of two rectangles to enclose.
`src2`	The second of two rectangles to enclose.
`dstRect`	The rectangle that encloses them.

DESCRIPTION

The `UnionRect` procedure returns in the `dstRect` parameter the smallest rectangle that encloses both of the rectangles you specify in the `src1` and `src2` parameters. One of the source rectangles may also be the destination.

PtInRect

To determine whether a pixel below is enclosed in a rectangle, use the `PtInRect` function.

```
FUNCTION PtInRect (pt: Point; r: Rect): Boolean;
```

`pt`	The point to test.
`r`	The rectangle to test.

DESCRIPTION

The `PtInRect` function determines whether the pixel below and to the right of the point you specify in the `pt` parameter is enclosed in the rectangle that you specify in the `Rect` parameter. The `PtInRect` function returns `TRUE` if it is, `FALSE` if it is not.

Pt2Rect

To determine the smallest rectangle that encloses two given points, use the `Pt2Rect` procedure.

```
PROCEDURE Pt2Rect (pt1,pt2: Point; VAR dstRect: Rect);
```

`pt1`	The first of two points to enclose.
`pt2`	The second of two points to enclose.
`dstRect`	The smallest rectangle that can enclose them.

DESCRIPTION

The `Pt2Rect` procedure returns in the `dstRect` parameter the smallest rectangle that encloses the two points you specify in the `pt1` and `pt2` parameters.

PtToAngle

To calculate an angle between a vertical line pointing straight up from the center of a rectangle and a line from the center to a given point, use the `PtToAngle` procedure.

```
PROCEDURE PtToAngle (r: Rect; pt: Point; VAR angle: Integer);
```

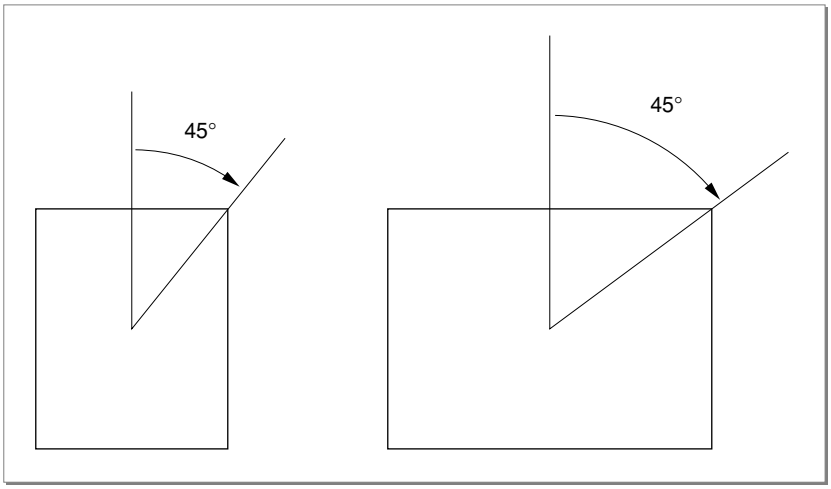
`r`	The rectangle to examine.
`pt`	The point to which an angle is to be calculated.
`angle`	The resulting angle.

DESCRIPTION

The `PtToAngle` procedure returns in the `angle` parameter the angle between a vertical line (pointing straight up from the center of the rectangle that you specify in the `r` parameter) and a line from the center of that rectangle to a point (which you specify in the `pt` parameter).

The result returned in the `angle` parameter is specified in degrees from 0 to 359, measured clockwise from 12 o'clock, with 90° at 3 o'clock, 180° at 6 o'clock, and 270° at 9 o'clock. Other angles are measured relative to the rectangle. If the line to the given point goes through the upper-right corner of the rectangle, the angle returned is 45°, even if the rectangle isn't square; if it goes through the lower-right corner, the angle is 135°, and so on, as shown in Figure 3-18.

Figure 3-18 Forty-five-degree angles as returned by the `PtToAngle` procedure



The angle returned might be used as input to one of the procedures that manipulate arcs and wedges, as described in "Drawing Arcs and Wedges" beginning on page 3-71.

EqualRect

To determine whether two rectangles are equal, you can use the `EqualRect` function.

```
FUNCTION EqualRect (rect1,rect2: Rect): Boolean;
```

`rect1` The first of two rectangles to compare.

`rect2` The second of two rectangles to compare.

DESCRIPTION

The `EqualRect` function compares the rectangles you specify in the `rect1` and `rect2` parameters and returns `TRUE` if they're equal, `FALSE` if they're not.

EmptyRect

To determine whether a rectangle is an empty rectangle, use the `EmptyRect` function.

```
FUNCTION EmptyRect (r: Rect): Boolean;
```

`r` The rectangle to examine.

DESCRIPTION

The `EmptyRect` function returns `TRUE` if the rectangle that you specify in the `r` parameter is an empty rectangle, `FALSE` if it is not. A rectangle is considered empty if the bottom coordinate is less than or equal to the top coordinate or if the right coordinate is less than or equal to the left.

Drawing Rectangles

A rectangle is defined by a data structure of type `Rect`, in which you specify two points (for the upper-left and lower-right corners of the rectangle) or four boundary coordinates (one for each side of the rectangle). After defining a rectangle (such as by using the `SetRect` procedure), you can use the `FrameRect` procedure to outline it with the size, pattern, and pattern mode of the graphics pen.

You can use the `PaintRect` procedure to draw a rectangle's interior with the pattern of the graphics pen, using the pattern mode of the graphics pen.

Using the `FillRect` procedure, you can draw a rectangle's interior with any pattern you specify. The procedure transfers the pattern with the `patCopy` pattern mode, which directly copies your requested pattern into the shape.

You can use the `EraseRect` procedure to erase a rectangle; this procedure fills the rectangle's interior with the background pattern for the current graphics port. Making the shape blend into the background pattern of the graphics port effectively erases the shape. For example, you can use `EraseRect` to erase the port rectangle for a window before redrawing into the window.

You can use the `InvertRect` procedure to invert a rectangle; this procedure reverses the colors of all pixels within the rectangle's boundary. On a black-and-white monitor, this changes all black pixels in the shape to white, and changes all white pixels to black. Although this procedure operates on color pixels in color graphics ports, the results are predictable only with direct pixels or 1-bit pixel maps.

FrameRect

To draw an outline inside a rectangle, use the `FrameRect` procedure.

```
PROCEDURE FrameRect (r: Rect);
```

`r` The rectangle to frame.

DESCRIPTION

Using the pattern, pattern mode, and size of the graphics pen for the current graphics port, the `FrameRect` procedure draws an outline just inside the rectangle that you specify in the `r` parameter. The outline is as wide as the pen width and as tall as the pen height. The pen location does not change.

If a region is open and being formed, the outside outline of the new rectangle is mathematically added to the region's boundary.

SPECIAL CONSIDERATIONS

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

SEE ALSO

Listing 3-4 on page 3-23 illustrates how to use this procedure.

PaintRect

To paint a rectangle with the graphics pen's pattern and pattern mode, use the `PaintRect` procedure.

```
PROCEDURE PaintRect (r: Rect);
```

`r` The rectangle to paint.

DESCRIPTION

The `PaintRect` procedure draws the interior of the rectangle that you specify in the `r` parameter with the pen pattern for the current graphics port, according to the pattern mode for the current graphics port. The pen location does not change.

SPECIAL CONSIDERATIONS

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

SEE ALSO

Listing 3-5 on page 3-24 illustrates how to use this procedure.

You can use the `FillRect` procedure, described next, to draw the interior of a rectangle with a pen pattern different from that for the current graphics port.

FillRect

To fill a rectangle with any available bit pattern, use the `FillRect` procedure.

```
PROCEDURE FillRect (r: Rect; pat: Pattern);
```

`r` The rectangle to fill.

`pat` The bit pattern to use for the fill. Figure 3-3 on page 3-7 illustrates the default fill patterns and the constants you can use to represent them.

DESCRIPTION

Using the `patCopy` pattern mode, the `FillRect` procedure draws the interior of the rectangle that you specify in the `r` parameter with the pattern defined in the `Pattern` record that you specify in the `pat` parameter. This procedure leaves the pen location unchanged.

SPECIAL CONSIDERATIONS

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

SEE ALSO

Listing 3-5 on page 3-24 illustrates how to use this procedure.

The `patCopy` pattern mode is described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8. The `Pattern` record is described on page 3-40. You can use the `GetPattern` and `GetIndPattern` routines, described on page 3-126 and page 3-127, respectively, to get a pattern stored in a resource.

You can use the `PaintRect` procedure, described in the previous section, to draw the interior of a rectangle with the pen pattern for the current graphics port. To fill a rectangle with a pixel pattern, use the `FillCRect` procedure, which is described in the chapter "Color QuickDraw."

EraseRect

To erase a rectangle, use the `EraseRect` procedure.

```
PROCEDURE EraseRect (r: Rect);
```

`r` The rectangle to erase.

DESCRIPTION

Using the `patCopy` pattern mode, the `EraseRect` procedure draws the interior of the rectangle that you specify in the `r` parameter with the background pattern for the current graphics port. This effectively erases the rectangle specified in the `r` parameter. For example, you can use `EraseRect` to erase the port rectangle for a window before redrawing into the window.

This procedure leaves the location of the graphics pen unchanged.

SPECIAL CONSIDERATIONS

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

SEE ALSO

Listing 6-2 on page 6-10 in the chapter "Offscreen Graphics Worlds" in this book illustrates how to use `EraseRect` to initialize an offscreen pixel map. The `patCopy` pattern mode is described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8.

InvertRect

To invert the pixels enclosed by a rectangle, use the `InvertRect` procedure.

```
PROCEDURE InvertRect (r: Rect);
```

`r` The rectangle whose enclosed pixels are to be inverted.

DESCRIPTION

The `InvertRect` procedure inverts the pixels enclosed by the rectangle that you specify in the `r` parameter. Every white pixel becomes black and every black pixel becomes white. The pen location does not change.

SPECIAL CONSIDERATIONS

The `InvertRect` procedure was designed for 1-bit images in basic graphics ports. This procedure operates on color pixels in color graphics ports, but the results are predictable only with direct pixels or 1-bit pixel maps. For indexed pixels, Color QuickDraw performs the inversion on the pixel indexes, which means the results depend entirely on the contents of the CLUT (which is described in the chapter "Color QuickDraw"). The eight colors used in basic QuickDraw are stored in a color table represented by the global variable `QDColors`. To display those eight basic QuickDraw colors on an indexed device, Color QuickDraw uses the Color Manager to obtain indexes to the colors in the CLUT that best map to the colors in the `QDColors` color table. Because the index, not the color value, is inverted, the results are unpredictable.

Inversion works better for direct pixels. Inverting a pure green, for example, that has red, green, and blue component values of $0000, $FFFF, and $0000 results in magenta, which has component values of $FFFF, $0000, and $FFFF.

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

Drawing Rounded Rectangles

As with rectangles, QuickDraw provides routines with which you can frame, paint, fill, erase, and invert rounded rectangles. Rounded rectangles are rectangles with rounded corners defined by the width and height of the ovals forming their corners.

You can use the `FrameRoundRect` procedure to draw an outline of a rounded rectangle with the size, pattern, and pattern mode of the graphics pen. You can use the `PaintRoundRect` procedure to draw a rounded rectangle's interior with the pattern of the graphics pen, using the pattern mode of the graphics pen.

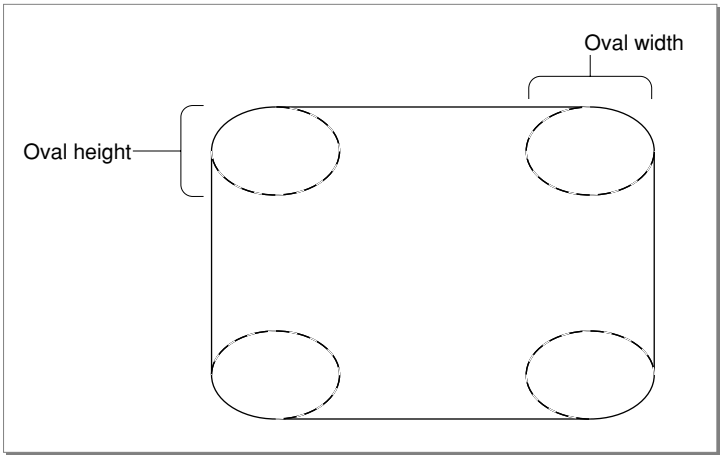
Using the `FillRoundRect` procedure, you can draw a rounded rectangle's interior with any pattern you specify. The procedure transfers the pattern with the `patCopy` pattern mode, which directly copies your requested pattern into the shape.

You can use the `EraseRoundRect` procedure to erase a rounded rectangle; this procedure fills the rectangle's interior with the background pattern for the current graphics port.

You can use the `InvertRoundRect` procedure to invert a rounded rectangle; this procedure reverses the colors of all pixels within the rounded rectangle. Although this procedure operates on color pixels in color graphics ports, the results are predictable only with 1-bit and direct color pixels.

When using these procedures, you specify a rectangle, which is defined by a data structure of type `Rect`. You must also specify the width and height of the ovals that describe the curvature of the rounded corners, as shown in Figure 3-19.

Figure 3-19 Oval width and height in rounded rectangles



FrameRoundRect

To draw an outline inside a rounded rectangle, use the `FrameRoundRect` procedure.

```
PROCEDURE FrameRoundRect (r: Rect; ovalWidth,ovalHeight: Integer);
```

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

`ovalWidth` The width of the oval defining the rounded corner.

`ovalHeight` The height of the oval defining the rounded corner.

DESCRIPTION

Using the pattern, pattern mode, and size of the graphics pen for the current graphics port, the `FrameRoundRect` procedure draws an outline just inside the rounded rectangle bounded by the rectangle that you specify in the `r` parameter. The outline is as wide as the pen width and as tall as the pen height. The pen location does not change.

Use the `ovalWidth` and `ovalHeight` parameters to specify the diameters of curvature for the corners of the rounded rectangle.

If a region is open and being formed, the outside outline of the new rounded rectangle is mathematically added to the region's boundary.

SPECIAL CONSIDERATIONS

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

PaintRoundRect

To paint a rounded rectangle with the graphics pen's pattern and pattern mode, use the `PaintRoundRect` procedure.

```
PROCEDURE PaintRoundRect (r: Rect; ovalWidth,ovalHeight: Integer);
```

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

`ovalWidth` The width of the oval defining the rounded corner.

`ovalHeight` The height of the oval defining the rounded corner.

DESCRIPTION

Using the pattern and pattern mode of the graphics pen for the current graphics port, the `PaintRoundRect` procedure draws the interior of the rounded rectangle bounded by the rectangle that you specify in the `r` parameter. Use the `ovalWidth` and `ovalHeight` parameters to specify the diameters of curvature for the corners of the rounded rectangle.

The pen location does not change.

SPECIAL CONSIDERATIONS

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

SEE ALSO

You can use the `FillRoundRect` procedure, described next, to draw the interior of a rounded rectangle with a pen pattern different from that for the current graphics port.

FillRoundRect

To fill a rounded rectangle with any available bit pattern, use the `FillRoundRect` procedure.

```
PROCEDURE FillRoundRect (r: Rect; ovalWidth,ovalHeight: Integer;
                         pat: Pattern);
```

`r`	The rectangle that defines the rounded rectangle's boundaries.
`ovalWidth`	The width of the oval defining the rounded corner.
`ovalHeight`	The height of the oval defining the rounded corner.
`pat`	The bit pattern to use for the fill. Figure 3-3 on page 3-7 illustrates the default fill patterns and the constants you can use to represent them.

DESCRIPTION

Using the `patCopy` pattern mode, the `FillRoundRect` procedure draws the interior of the rounded rectangle bounded by the rectangle that you specify in the `r` parameter with the bit pattern defined in the `Pattern` record that you specify in the `pat` parameter. Use the `ovalWidth` and `ovalHeight` parameters to specify the diameters of curvature for the corners. The pen location does not change.

SPECIAL CONSIDERATIONS

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

SEE ALSO

You can use the `GetPattern` and `GetIndPattern` routines, described on page 3-126 and page 3-127, respectively, to get a pattern stored in a resource. The `patCopy` pattern mode is described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8. The `Pattern` record is described on page 3-40.

You can use the `PaintRoundRect` procedure, described in the previous section, to draw the interior of a rounded rectangle with the pen pattern for the current graphics port. To fill a rounded rectangle with a pixel pattern, use the `FillCRoundRect` procedure, which is described in the chapter "Color QuickDraw."

EraseRoundRect

To erase a rounded rectangle, use the `EraseRoundRect` procedure.

```
PROCEDURE EraseRoundRect (r: Rect; ovalWidth,ovalHeight: Integer);
```

`r`	The rectangle that defines the rounded rectangle's boundaries.
`ovalWidth`	The width of the oval defining the rounded corner.
`ovalHeight`	The height of the oval defining the rounded corner.

DESCRIPTION

Using the `patCopy` pattern mode, the `EraseRoundRect` procedure draws the interior of the rounded rectangle bounded by the rectangle that you specify in the `r` parameter with the background pattern of the current graphics port. This effectively erases the rounded rectangle. Use the `ovalWidth` and `ovalHeight` parameters to specify the diameters of curvature for the corners of the rounded rectangle.

This procedure leaves the location of the graphics pen unchanged.

SPECIAL CONSIDERATIONS

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

SEE ALSO

The `patCopy` pattern mode is described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8.

InvertRoundRect

To invert the pixels enclosed by a rounded rectangle, use the `InvertRoundRect` procedure.

```
PROCEDURE InvertRoundRect (r: Rect;
                           ovalWidth,
                           ovalHeight: Integer);
```

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

`ovalWidth` The width of the oval defining the rounded corner.

`ovalHeight` The height of the oval defining the rounded corner.

DESCRIPTION

The `InvertRoundRect` procedure inverts the pixels enclosed by the rounded rectangle bounded by the rectangle that you specify in the `r` parameter. Every white pixel becomes black and every black pixel becomes white. The `ovalWidth` and `ovalHeight` parameters specify the diameters of curvature for the corners. The pen location does not change.

SPECIAL CONSIDERATIONS

The `InvertRoundRect` procedure was designed for 1-bit images in basic graphics ports. This procedure operates on color pixels in color graphics ports, but the results are predictable only with direct devices or 1-bit pixel maps. For indexed pixels, Color QuickDraw performs the inversion on the pixel indexes, which means the results depend entirely on the contents of the CLUT (which is described in the chapter "Color QuickDraw"). The eight colors used in basic QuickDraw are stored in a color table represented by the global variable `QDColors`. To display those eight basic QuickDraw colors on an indexed device, Color QuickDraw uses the Color Manager to obtain indexes to the colors in the CLUT that best map to the colors in the `QDColors` color table. Because the index, not the color value, is inverted, the results are unpredictable.

Inversion works better for direct pixels. Inverting a pure green, for example, that has red, green, and blue component values of $0000, $FFFF, and $0000 results in magenta, which has component values of $FFFF, $0000, and $FFFF.

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

Drawing Ovals

An oval is a circular or elliptical shape defined by the bounding rectangle that encloses it. You can use the `FrameOval` procedure to draw its outline, or the `PaintOval` or `FillOval` procedure to draw its interior with a pattern. You can use the `EraseOval` procedure to erase an oval, and you can use the `InvertOval` procedure to reverse the colors of all pixels within the oval. (Although this procedure operates on color pixels in color graphics ports, the results of `InvertOval` are predictable only with 1-bit or direct color pixels.)

FrameOval

To draw an outline inside an oval, use the `FrameOval` procedure.

```
PROCEDURE FrameOval (r: Rect);
```

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

DESCRIPTION

Using the pattern, pattern mode, and size of the graphics pen for the current graphics port, the `FrameOval` procedure draws an outline just inside the oval with the bounding rectangle that you specify in the `r` parameter. The outline is as wide as the pen width and as tall as the pen height. The pen location does not change.

If a region is open and being formed, the outside outline of the new oval is mathematically added to the region's boundary.

SPECIAL CONSIDERATIONS

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

SEE ALSO

Listing 3-6 on page 3-25 illustrates how to use this procedure.

PaintOval

To paint an oval with the graphics pen's pattern and pattern mode, use the `PaintOval` procedure.

```
PROCEDURE PaintOval (r: Rect);
```

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

DESCRIPTION

Using the pen pattern and pattern mode for the current graphics port, the `PaintOval` procedure draws the interior of an oval just inside the bounding rectangle that you specify in the `r` parameter. The pen location does not change.

SPECIAL CONSIDERATIONS

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

SEE ALSO

You can use the `FillOval` procedure, described next, to draw the interior of an oval with a pen pattern different from that for the current graphics port.

FillOval

To fill an oval with any available bit pattern, use the `FillOval` procedure.

```
PROCEDURE FillOval (r: Rect; pat: Pattern);
```

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

`pat` The bit pattern to use for the fill. Figure 3-3 on page 3-7 illustrates the default fill patterns and the constants you can use to represent them.

DESCRIPTION

Using the `patCopy` pattern mode and the bit pattern defined in the `Pattern` record that you specify in the `pat` parameter, the `FillOval` procedure draws the interior of an oval just inside the bounding rectangle that you specify in the `r` parameter. The pen location does not change.

SPECIAL CONSIDERATIONS

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

SEE ALSO

You can use the `GetPattern` and `GetIndPattern` routines, described on page 3-126 and page 3-127, respecively, to get a pattern stored in a resource. The `patCopy` pattern mode is described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8. The `Pattern` record is described on page 3-40.

You can use the `PaintOval` procedure, described in the previous section, to draw the interior of an oval with the pen pattern for the current graphics port. To fill an oval with a pixel pattern, use the `FillCOval` procedure, which is described in the chapter "Color QuickDraw."

EraseOval

To erase an oval, use the `EraseOval` procedure.

```
PROCEDURE EraseOval (r: Rect);
```

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

DESCRIPTION

Using the background pattern for the current graphics port and the `patCopy` pattern mode, the `EraseOval` procedure draws the interior of an oval just inside the bounding rectangle that you specify in the `r` parameter. This effectively erases the oval bounded by the specified rectangle.

This procedure leaves the location of the graphics pen unchanged.

SPECIAL CONSIDERATIONS

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

SEE ALSO

The `patCopy` pattern mode is described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8.

InvertOval

To invert the pixels enclosed by an oval, use the `InvertOval` procedure.

```
PROCEDURE InvertOval (r: Rect);
```

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

DESCRIPTION

The `InvertOval` procedure inverts the pixels enclosed by an oval just inside the bounding rectangle that you specify in the `r` parameter. Every white pixel becomes black and every black pixel becomes white. The pen location does not change.

SPECIAL CONSIDERATIONS

The `InvertOval` procedure was designed for 1-bit images in basic graphics ports. This procedure operates on color pixels in color graphics ports, but the results are predictable only with direct devices or 1-bit pixel maps. For indexed pixels, Color QuickDraw performs the inversion on the pixel indexes, which means the results depend entirely on the contents of the CLUT (which is described in the chapter "Color QuickDraw"). The eight colors used in basic QuickDraw are stored in a color table represented by the global variable `QDColors`. To display those eight basic QuickDraw colors on an indexed device, Color QuickDraw uses the Color Manager to obtain indexes to the colors in the CLUT that best map to the colors in the `QDColors` color table. Because the index, not the color value, is inverted, the results are unpredictable.

Inversion works better for direct pixels. Inverting a pure green, for example, that has red, green, and blue component values of $0000, $FFFF, and $0000 results in magenta, which has component values of $FFFF, $0000, and $FFFF.

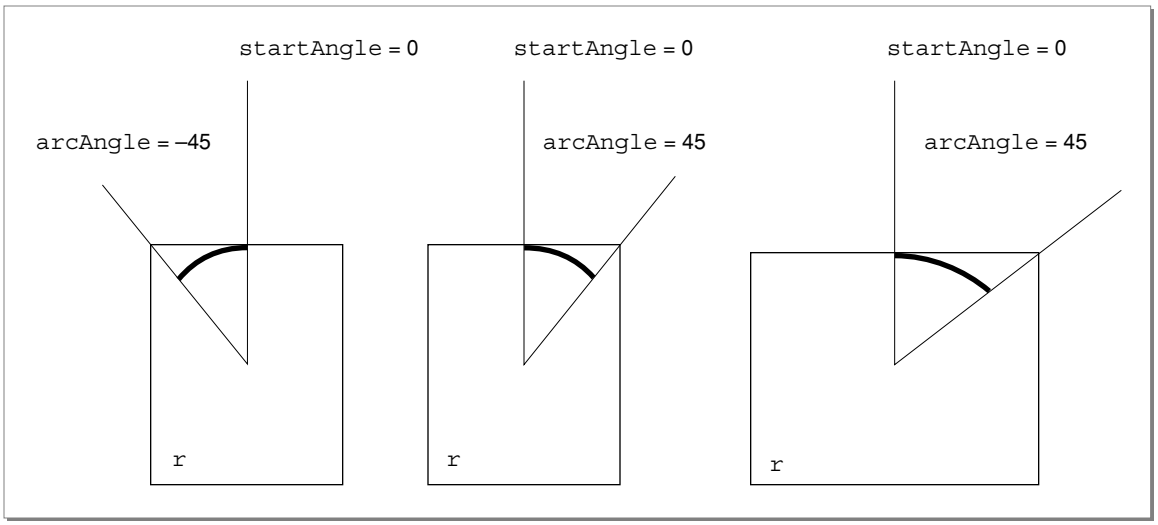
The `InvertOval` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

Drawing Arcs and Wedges

An arc is defined as a portion of an oval's circumference bounded by a pair of radii. A wedge is a pie-shaped segment bounded by a pair of radii, and it extends from the center of the oval to the circumference. You use the `FrameArc` procedure to draw an arc, and you use the `PaintArc` or `FillArc` procedure to draw a wedge. Using the `EraseArc` procedure, you can erase a wedge, and, using `InvertArc`, you can reverse the colors of all pixels within a wedge. (Although this procedure operates on color pixels in color graphics ports, the results of `InvertArc` are predictable only with 1-bit and direct color pixels.)

These procedures take three parameters: a rectangle that defines an oval's boundaries, an angle indicating the start of the arc (the variable `startAngle`), and an angle indicating the arc's extent (the variable `arcAngle`). For the angle parameters, 0° indicates a vertical line straight up from the center of the oval. Positive values indicate angles in the clockwise direction from this vertical line, and negative values indicate angles in the counterclockwise direction, as shown in Figure 3-20.

Figure 3-20 Using angles to define the radii for arcs and wedges



FrameArc

To draw an arc of the oval that fits inside a rectangle, use the `FrameArc` procedure.

```
PROCEDURE FrameArc (r: Rect; startAngle,arcAngle: Integer);
```

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

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

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

DESCRIPTION

Using the pattern, pattern mode, and size of the graphics pen for the current graphics port, the `FrameArc` procedure draws an arc of the oval bounded by the rectangle that you specify in the `r` parameter. Use the `startAngle` parameter to specify where the arc begins as modulo 360. Use the `arcAngle` parameter to specify how many degrees the arc covers. Specify whether the angles are in positive or negative degrees; a positive angle goes clockwise, while a negative angle goes counterclockwise. Zero degrees is at 12 o'clock high, 90° (or –270°) is at 3 o'clock, 180° (or –180°) is at 6 o'clock, and 270° (or –90°) is at 9 o'clock. Measure other angles relative to the bounding rectangle.

A line from the center of the rectangle through its upper-right corner is at 45°, even if the rectangle isn't square; a line through the lower-right corner is at 135°, and so on, as shown in Figure 3-20.

The arc is as wide as the pen width and as tall as the pen height. The pen location does not change.

SPECIAL CONSIDERATIONS

The `FrameArc` procedure differs from other QuickDraw procedures that frame shapes in that the arc is not mathematically added to the boundary of a region that's open and being formed.

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

SEE ALSO

Listing 3-7 on page 3-26 illustrates how to use this procedure.

PaintArc

To paint a wedge of the oval that fits inside a rectangle with the graphics pen's pattern and pattern mode, use the `PaintArc` procedure.

```
PROCEDURE PaintArc (r: Rect; startAngle,arcAngle: Integer);
```

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

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

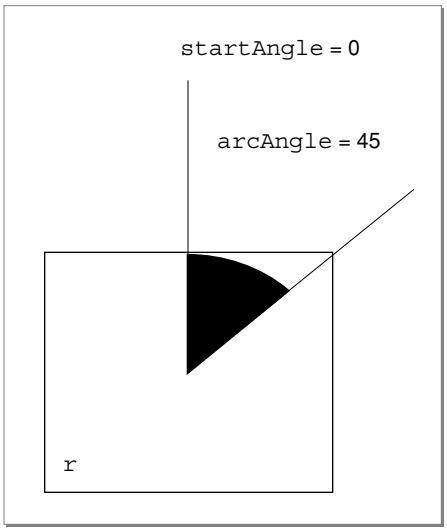
`arcAngle` The angle indicating the arc's extent.

DESCRIPTION

Using the pen pattern and pattern mode of the current graphics port, the `PaintArc` procedure draws a wedge of the oval bounded by the rectangle that you specify in the `r` parameter. As in the `FrameArc` procedure described in the previous section and illustrated in Figure 3-21, use the `startAngle` and `arcAngle` parameters to define the arc of the wedge.

The pen location does not change.

Figure 3-21 Using `PaintArc` to paint a 45° angle



SPECIAL CONSIDERATIONS

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

SEE ALSO

Listing 3-7 on page 3-26 illustrates how to use this procedure.

You can use the `FillArc` procedure, described next, to draw a wedge with a pattern different from that specified in the `pnPat` field of the current graphics port.

FillArc

To fill a wedge with any available bit pattern, use the `FillArc` procedure.

```
PROCEDURE FillArc (r: Rect; startAngle,arcAngle: Integer;
                   pat: Pattern);
```

`r`	The rectangle that defines an oval's boundaries.
`startAngle`	The angle indicating the start of the arc.
`arcAngle`	The angle indicating the arc's extent.
`pat`	The bit pattern to use for the fill. Figure 3-3 on page 3-7 illustrates the default fill patterns and the constants you can use to represent them.

DESCRIPTION

Using the `patCopy` pattern mode and the pattern defined in the `Pattern` record that you specify in the `pat` parameter, the `FillArc` procedure draws a wedge of the oval bounded by the rectangle that you specify in the `r` parameter. As in the `FrameArc` procedure described on page 3-72 and as illustrated in Figure 3-21, use the `startAngle` and `arcAngle` parameters to define the arc of the wedge.

This procedure leaves the location of the graphics pen unchanged.

SPECIAL CONSIDERATIONS

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

SEE ALSO

You can use the `GetPattern` and `GetIndPattern` routines, described on page 3-126 and page 3-127, respectively, to get a pattern stored in a resource. The `patCopy` pattern mode is described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8. The `Pattern` record is described on page 3-40.

You can use the `PaintArc` procedure, described in the previous section, to draw a wedge with the pen pattern for the current graphics port. To fill a wedge with a pixel pattern, use the `FillCArc` procedure, which is described in the chapter "Color QuickDraw."

EraseArc

To erase a wedge, use the `EraseArc` procedure.

```
PROCEDURE EraseArc (r: Rect; startAngle,arcAngle: Integer);
```

`r`	The rectangle that defines an oval's boundaries.
`startAngle`	The angle indicating the start of the arc.
`arcAngle`	The angle indicating the arc's extent.

DESCRIPTION

Using the `patCopy` pattern mode, the `EraseArc` procedure draws a wedge of the oval bounded by the rectangle that you specify in the `r` parameter with the background pattern for the current graphics port. As in the `FrameArc` procedure described on page 3-72 and as illustrated in Figure 3-21, use the `startAngle` and `arcAngle` parameters to define the arc of the wedge.

This procedure leaves the location of the graphics pen unchanged.

SPECIAL CONSIDERATIONS

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

SEE ALSO

The `patCopy` pattern mode is described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8.

InvertArc

To invert the pixels of a wedge, use the `InvertArc` procedure.

```
PROCEDURE InvertArc (r: Rect; startAngle,arcAngle: Integer);
```

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

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

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

DESCRIPTION

The `InvertArc` procedure inverts the pixels enclosed by a wedge of the oval bounded by the rectangle that you specify in the `r` parameter. Every white pixel becomes black and every black pixel becomes white. As in the `FrameArc` procedure described on page 3-72 and as illustrated in Figure 3-21, use the `startAngle` and `arcAngle` parameters to define the arc of the wedge.

This procedure leaves the location of the graphics pen unchanged.

SPECIAL CONSIDERATIONS

The `InvertArc` procedure was designed for 1-bit images in basic graphics ports. This procedure operates on color pixels in color graphics ports, but the results are predictable only with direct devices or 1-bit pixel maps. For indexed pixels, Color QuickDraw performs the inversion on the pixel indexes, which means the results depend entirely on the contents of the CLUT (which is described in the chapter "Color QuickDraw"). The eight colors used in basic QuickDraw are stored in a color table represented by the global variable `QDColors`. To display those eight basic QuickDraw colors on an indexed device, Color QuickDraw uses the Color Manager to obtain indexes to the colors in the CLUT that best map to the colors in the `QDColors` color table. Because the index, not the color value, is inverted, the results are unpredictable.

Inversion works better for direct pixels. Inverting a pure green, for example, that has red, green, and blue component values of $0000, $FFFF, and $0000 results in magenta, which has component values of $FFFF, $0000, and $FFFF.

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

Creating and Managing Polygons

A polygon is defined by a sequence of connected lines. To create a polygon, you first call the `OpenPoly` function and then some number of `LineTo` procedures to draw lines from the first vertex of the polygon to the second, from the second to the third, and so on, until you've drawn a line to the last vertex. You then use the `ClosePoly` procedure, which completes the figure by drawing a connecting line from the last vertex back to the first.

After you use the `OpenPoly` function to create a polygon, QuickDraw begins collecting the line-drawing information you provide into a `Polygon` record. The `OpenPoly` function returns a handle to the newly allocated `Polygon` record.

After defining a polygon in this way, you can draw it with the `FramePoly`, `PaintPoly`, and `FillPoly` procedures. You can move it by using the `OffSetPoly` procedure. When you are finished using the polygon, use the `KillPoly` procedure to release its memory. When using the `ClosePoly`, `OffsetPoly`, and `KillPoly` procedures, you refer to a polygon by the handle returned by `OpenPoly` when you first created the polygon.

Note
If, while your application draws a polygon, it exceeds available stack space in Color QuickDraw, the `QDError` function (described in the chapter "Color QuickDraw" in this book) returns the result code –144. ◆

OpenPoly

To begin defining a polygon, use the `OpenPoly` function.

```
FUNCTION OpenPoly:  PolyHandle;
```

DESCRIPTION

The `OpenPoly` function returns a handle to a new polygon and starts saving lines for processing as a polygon definition. While a polygon is open, all calls to the `Line` and `LineTo` procedures affect the outline of the polygon. Only the line endpoints affect the polygon definition; the pattern mode, pattern, and size do not affect it. The `OpenPoly` function calls the `HidePen` procedure, so no drawing occurs on the screen while the polygon is open (unless you call the `ShowPen` procedure just after calling `OpenPoly`, or you called `ShowPen` previously without balancing it by a call to `HidePen`).

A polygon should consist of a sequence of connected lines. The `OpenPoly` function stores the definition for a polygon in a `Polygon` record.

When a polygon is open, the current graphics port's `polySave` field contains a handle to information related to the polygon definition. If you want to temporarily disable the polygon definition, you can save the current value of this field, set the field to `NIL`, and later restore the saved value to resume the polygon definition.

Even though the onscreen presentation of a polygon is clipped, the definition of a polygon is not; you can define a polygon anywhere on the coordinate plane.

When you are finished calling the line-drawing routines that define your polygon, use the `ClosePoly` procedure, described next.

SPECIAL CONSIDERATIONS

Do not call `OpenPoly` while a region or another polygon is already open.

Polygons are limited to 64 KB. You can determine the polygon size while it's being formed by calling the Memory Manager function `GetHandleSize`, which is described in *Inside Macintosh: Memory*.

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

SEE ALSO

Listing 3-10 on page 3-30 illustrates how to use this function to create a triangle. The `Polygon` record is described on page 3-37.

ClosePoly

To complete the collection of lines that defines your polygon, use the `ClosePoly` procedure.

```
PROCEDURE ClosePoly;
```

DESCRIPTION

The `ClosePoly` procedure stops collecting line-drawing commands for the currently open polygon and computes the `polyBBox` field of the `Polygon` record. You should call `ClosePoly` only once for every call to the `OpenPoly` function.

The `ClosePoly` procedure uses the `ShowPen` procedure, balancing the call to the `HidePen` procedure made by the `OpenPoly` function.

SPECIAL CONSIDERATIONS

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

SEE ALSO

Listing 3-10 on page 3-30 illustrates how to use this procedure when creating a triangle.

OffsetPoly

To move a polygon, use the `OffsetPoly` procedure.

```
PROCEDURE OffsetPoly (poly: PolyHandle; dh,dv: Integer);
```

`poly`	A handle to a polygon to move.
`dh`	The horizontal distance to move the polygon.
`dv`	The vertical distance to move the polygon.

DESCRIPTION

The `OffsetPoly` procedure moves the polygon whose handle you pass in the `poly` parameter by adding the value you specify in the `dh` parameter to the horizontal coordinates of its points, and by adding the value you specify in the `dv` parameter to the vertical coordinates of all points of its region boundary. If the values of `dh` and `dv` are positive, the movement is to the right and down; if either is negative, the corresponding movement is in the opposite direction. The region retains its size and shape. This doesn't affect the screen unless you subsequently call a routine to draw the region.

Note
`OffsetPoly` is an especially efficient operation, because the data defining a polygon is stored relative to the first point of the polygon and so isn't actually changed by `OffsetPoly`. ◆

KillPoly

To release the memory occupied by a polygon, use the `KillPoly` procedure.

```
PROCEDURE KillPoly (poly: PolyHandle);
```

`poly`	A handle to the polygon to dispose of.

DESCRIPTION

The `KillPoly` procedure releases the memory used by the polygon whose handle you pass in the `poly` parameter. Use `KillPoly` only when you're completely through with a polygon.

SPECIAL CONSIDERATIONS

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

SEE ALSO

Listing 3-10 on page 3-30 illustrates how to use this procedure.

Drawing Polygons

After defining a polygon by using the `OpenPoly` function, a number of line-drawing procedures, and the `ClosePoly` procedure, you can draw the polygon's outline with the `FramePoly` procedure. You can draw its interior with the `PaintPoly` and `FillPoly` procedures. You can erase its interior by using the `ErasePoly` procedure, and you can use the `InvertPoly` procedure to reverse the colors of the pixels within it. In all of these procedures, you refer to a polygon by the handle returned by `OpenPoly` when you first created the polygon.

Four of these procedures—`PaintPoly`, `ErasePoly`, `InvertPoly`, and `FillPoly`—temporarily convert the polygon into a region to perform their operations. The amount of memory required for this temporary region may be far greater than the amount required by the polygon alone.

You can estimate the size of this region by scaling down the polygon with the `MapPoly` procedure (described on page 3-108), converting the polygon into a region, checking the region's size with the Memory Manager function `GetHandleSize`, and multiplying that value by the factor by which you scaled the polygon.

▲ **WARNING**
The results of these graphics operations are undefined whenever any horizontal or vertical line drawn through the polygon would intersect the polygon's outline more than 50 times. ▲

FramePoly

To draw the outline of a polygon, use the `FramePoly` procedure.

```
PROCEDURE FramePoly (poly:  PolyHandle);
```

`poly` A handle to the polygon to draw.

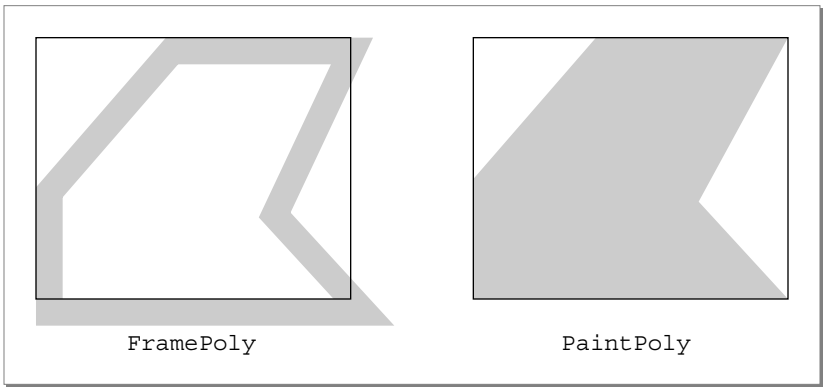
DESCRIPTION

Using the current graphics port's pen pattern, pattern mode, and size, the `FramePoly` procedure plays back the line-drawing commands that define the polygon whose handle you pass in the `poly` parameter.

The graphics pen hangs below and to the right of each point on the boundary of the polygon. Thus, the drawn polygon extends beyond the right and bottom edges of the polygon's bounding rectangle (which is stored in the `polyBBox` field of the `Polygon` record) by the pen width and pen height, respectively. All other graphics

operations, such as painting a polygon with the `PaintPoly` procedure, occur strictly within the boundary of the polygon, as illustrated in Figure 3-22.

Figure 3-22 Framing and painting polygons



If a polygon is open and being formed, `FramePoly` affects the outline of the polygon just as if the line-drawing routines themselves had been called. If a region is open and being formed, the outside outline of the polygon being framed is mathematically added to the region's boundary.

SPECIAL CONSIDERATIONS

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

PaintPoly

To paint a polygon with the graphics pen's pattern and pattern mode, use the `PaintPoly` procedure.

```
PROCEDURE PaintPoly (poly: PolyHandle);
```

`poly` A handle to the polygon to paint.

DESCRIPTION

Using the pen pattern and pattern mode for the current graphics port, the `PaintPoly` procedure draws the interior of a polygon whose handle you pass in the `poly` parameter. The pen location does not change.

SPECIAL CONSIDERATIONS

Do not create a height or width for the polygon greater than 32,767 pixels, or `PaintPoly` will crash.

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

SEE ALSO

You can use the `FillPoly` procedure, described next, to draw the interior of a polygon with a pattern different from that specified in the `pnPat` field of the current graphics port.

FillPoly

To fill a polygon with any available bit pattern, use the `FillPoly` procedure.

```
PROCEDURE FillPoly (poly: PolyHandle; pat: Pattern);
```

`poly`	A handle to the polygon to fill.
`pat`	The bit pattern to use for the fill. Figure 3-3 on page 3-7 illustrates the default fill patterns and the constants you can use to represent them.

DESCRIPTION

Using the `patCopy` pattern mode, the `FillPoly` procedure draws the interior of the polygon whose handle you pass in the `poly` parameter with the pattern defined in the `Pattern` record that you specify in the `pat` parameter.

This procedure leaves the location of the graphics pen unchanged.

SPECIAL CONSIDERATIONS

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

SEE ALSO

Listing 3-10 on page 3-30 illustrates how to use this procedure to fill a triangle.

You can use the `GetPattern` and `GetIndPattern` routines, described on page 3-126 and page 3-127, respectively, to get a pattern stored in a resource. The `patCopy` pattern mode is described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8. The `Pattern` record is described on page 3-40.

You can use the `PaintPoly` procedure, described in the previous section, to draw the interior of a polygon with the pen pattern for the current graphics port. To fill a polygon with a pixel pattern, use the `FillCPoly` procedure, which is described in the chapter "Color QuickDraw."

ErasePoly

To erase a polygon, use the `ErasePoly` procedure.

```
PROCEDURE ErasePoly (poly: PolyHandle);
```

`poly` A handle to the polygon to erase.

DESCRIPTION

Using the `patCopy` pattern mode, the `ErasePoly` procedure draws the interior of the polygon whose handle you pass in the `poly` parameter with the background pattern for the current graphics port.

This procedure leaves the location of the graphics pen unchanged.

SPECIAL CONSIDERATIONS

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

SEE ALSO

The `patCopy` pattern mode is described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8.

InvertPoly

To invert the pixels enclosed by a polygon, use the `InvertPoly` procedure.

```
PROCEDURE InvertPoly (poly: PolyHandle);
```

`poly` A handle to a polygon, the pixels of which you want to invert.

DESCRIPTION

The `InvertPoly` procedure inverts the pixels enclosed by the polygon whose handle you pass in the `poly` parameter. Every white pixel becomes black and every black pixel becomes white.

This procedure leaves the location of the graphics pen unchanged.

SPECIAL CONSIDERATIONS

The `InvertPoly` procedure was designed for 1-bit images in basic graphics ports. This procedure operates on color pixels in color graphics ports, but the results are predictable only with 1-bit or direct pixels. For indexed pixels, Color QuickDraw performs the inversion on the pixel indexes, which means the results depend entirely on the contents of the CLUT (which is described in the chapter "Color QuickDraw"). The eight colors used in basic QuickDraw are stored in a color table represented by the global variable `QDColors`. To display those eight basic QuickDraw colors on an indexed device, Color QuickDraw uses the Color Manager to obtain indexes to the colors in the CLUT that best map to the colors in the `QDColors` color table. Because the index, not the color value, is inverted, the results are unpredictable.

Inversion works better for direct pixels. Inverting a pure green, for example, that has red, green, and blue component values of $0000, $FFFF, and $0000 results in magenta, which has component values of $FFFF, $0000, and $FFFF.

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

Creating and Managing Regions

To define a region, you can use any set of lines or shapes, including other regions, so long as the region's outline consists of one or more closed loops. To begin defining a region, you must use the `NewRgn` function to allocate space for it, and then call the `OpenRgn` procedure. You can then use any QuickDraw routines to construct the outline of the region. When you are finished constructing the region, use the `CloseRgn` procedure.

The `NewRgn` function returns a handle to the newly allocated `Region` record. After you use the `OpenRgn` procedure, QuickDraw begins collecting the drawing information you provide into this `Region` record. (The `Region` record is described in the chapter "Basic QuickDraw.")

After defining a region in this way, you can display it with the `FrameRgn`, `PaintRgn`, and `FillRgn` procedures. When you are finished using the region, use the `DisposeRgn` procedure to release its memory.

You can use the `SetEmptyRgn` procedure to set a region to be empty, `SetRectRgn` to change it into a rectangle, `OffsetRgn` to move it, `InsetRgn` to shrink or expand it, `PtInRgn` to determine whether a pixel lies within it, `RectInRgn` to determine whether a rectangle intersects it, `EmptyRgn` to determine whether it is an empty region, and `CopyRgn` to make a copy of it. You can use the `RectRgn` procedure to make a region out of a rectangle. You can use the `SectRgn` procedure to calculate the intersection of two regions, `UnionRgn` to calculate the union of two regions, `DiffRgn` to subtract one region from another, `XorRgn` to calculate the difference between the union and the intersection of two regions, and `EqualRgn` to determine whether two regions have identical sizes, shapes, and locations.

When using these procedures, you refer to a region by the handle returned by `NewRgn` when you first allocated memory for the region.

▲ **WARNING**
Ensure that the memory for a region is valid before calling these routines to manipulate that region; if there isn't sufficient memory, the system may crash. Regions are limited to 32 KB in size in basic QuickDraw and 64 KB in Color QuickDraw. Before defining a region, you can use the Memory Manager function `MaxMem` to determine whether the memory for the region is valid. You can determine the current size of an existing region by calling the Memory Manager function `GetHandleSize`. (Both `MaxMem` and `GetHandleSize` are described in *Inside Macintosh: Memory*.) When you record drawing operations in an open region, the resulting region description may overflow the 32 KB or 64 KB limit. Should this happen in Color QuickDraw, the `QDError` function (described in the chapter "Color QuickDraw" in this book) returns the result code `regionTooBigError`. ▲

If the points or rectangles supplied to these routines are defined in a graphics port other than your current graphics port, you must convert them to the local coordinate system of your current graphics port. You can accomplish this by using the `SetPort` procedure to change to the graphics port containing the points or rectangles, using the `LocalGlobal` procedure to convert their locations to global coordinates, using `SetPort` to return to your starting graphics port, and then using the `GlobalToLocal` procedure to convert the locations of points or rectangles to the local coordinates of your current graphics port. These procedures are described in the chapter "Basic QuickDraw."

NewRgn

To begin creating a new region, use the `NewRgn` function.

```
FUNCTION NewRgn: RgnHandle;
```

DESCRIPTION

The `NewRgn` function allocates space for a new, variable-size region; initializes it to the empty region defined by the rectangle (0,0,0,0); and returns a handle to the new region. This is the only function that creates a new region; other routines merely alter the size or shape of existing regions.

To begin defining a region, use the `OpenRgn` procedure, described next.

SPECIAL CONSIDERATIONS

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

Use the Memory Manager function `MaxMem` (described in *Inside Macintosh: Memory*) to determine whether the memory for the region is valid before using `NewRgn`.

SEE ALSO

Listing 3-8 on page 3-28 and Listing 3-9 on page 3-29 illustrate how to use this function.

OpenRgn

To begin defining a region, use the `OpenRgn` procedure.

```
PROCEDURE OpenRgn;
```

DESCRIPTION

The `OpenRgn` procedure allocates temporary memory to start saving lines and framed shapes for processing as a region definition. Call `OpenRgn` only after initializing a region with the `NewRgn` function.

The `NewRgn` function stores the definition for a region in a `Region` record.

While a region is open, all calls to `Line`, `LineTo`, and the procedures that draw framed shapes (except arcs) affect the outline of the region. Only the line endpoints and shape boundaries affect the region definition—the pattern mode, pattern, and size do not affect it.

When you are finished defining the region, call the `CloseRgn` procedure.

The `OpenRgn` procedure calls `HidePen`, so no drawing occurs on the screen while the region is open (unless you call `ShowPen` just after `OpenRgn`, or you called `ShowPen` previously without balancing it by a call to `HidePen`). Since the pen hangs below and to the right of the pen location, drawing lines with even the smallest pen changes pixels that lie outside the region you define.

The outline of a region is mathematically defined and infinitely thin, and it separates the bit or pixel image into two groups of pixels: those within the region and those outside it.

A region should consist of one or more closed loops. Each framed shape itself constitutes a loop. Any lines drawn with the `Line` or `LineTo` procedure should connect with each other or with a framed shape. Even if the onscreen presentation of a region is clipped, the definition of a region is not; you can define a region anywhere on the coordinate plane with complete disregard for the location of various graphics port entities on that plane.

When a region is open, the current graphics port's `rgnSave` field contains a handle to information related to the region definition. If you want to temporarily disable the collection of lines and shapes, you can save the current value of this field, set the field to `NIL`, and later restore the saved value to resume the region definition. Also, calling `SetPort` while a region is being formed discontinues formation of the region until another call to `SetPort` resets the region's original graphics port.

SPECIAL CONSIDERATIONS

Regions are limited to 32 KB in size in basic QuickDraw and 64 KB in Color QuickDraw. You can determine the current size of an existing region by calling the Memory Manager function `GetHandleSize` (described in *Inside Macintosh: Memory*). When you record drawing operations in an open region, the resulting region description may overflow the 32 KB or 64 KB limit. Should this happen in Color QuickDraw, the `QDError` function (described in the chapter "Color QuickDraw" in this book) returns the result code `regionTooBigError`.

Do not call `OpenRgn` while another region or a polygon is already open. When you are finished constructing the region, use the `CloseRgn` procedure, which is described next.

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

SEE ALSO

Listing 3-8 on page 3-28 illustrates how to use this procedure. The `Region` record is described in the chapter "Basic QuickDraw."

CloseRgn

To organize a collection of lines and shapes into a region definition, use the `CloseRgn` procedure.

```
PROCEDURE CloseRgn (dstRgn: rgnHandle);
```

`dstRgn` The handle to the region to close.

DESCRIPTION

The `CloseRgn` procedure stops the collection of lines and framed shapes, organizes them into a region definition, and saves the result in the region whose handle you pass in the `dstRgn` parameter. The handle you pass in the `dstRgn` parameter should be a region handle returned by the `NewRgn` function.

The `CloseRgn` procedure does not create the destination region; you must have already allocated space for it by using the `OpenRgn` procedure. The `CloseRgn` procedure calls the `ShowPen` procedure, balancing the call to the `HidePen` procedure made by `OpenRgn`.

When you no longer need the memory occupied by the region, use the `DisposeRgn` procedure, described next.

SPECIAL CONSIDERATIONS

Regions are limited to 32 KB in size in basic QuickDraw and 64 KB in Color QuickDraw. When you record drawing operations in an open region, the resulting region description may overflow this limit. Should this happen in Color QuickDraw, the `QDError` function (described in the chapter "Color QuickDraw" in this book) returns the result code `regionTooBigError`. Since the resulting region is potentially corrupt, the `CloseRgn` procedure returns an empty region if it detects `QDError` has returned `regionTooBigError`.

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

SEE ALSO

Listing 3-8 on page 3-28 illustrates how to use this procedure.

DisposeRgn

To release the memory occupied by a region, use the `DisposeRgn` procedure.

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

`rgn` A handle to the region to dispose.

DESCRIPTION

The `DisposeRgn` procedure releases the memory occupied by the region whose handle you pass in the `rgn` parameter.

Use `DisposeRgn` only after you're completely through with a region.

SPECIAL CONSIDERATIONS

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

SEE ALSO

Listing 3-8 on page 3-28 and Listing 3-9 on page 3-29 illustrate how to use this procedure.

CopyRgn

To make a copy of a region, use the `CopyRgn` procedure.

```
PROCEDURE CopyRgn (srcRgn,dstRgn: RgnHandle);
```

`srcRgn` A handle to the region to copy.
`dstRgn` A handle to the region to receive the copy.

DESCRIPTION

The `CopyRgn` procedure copies the mathematical structure of the region whose handle you pass in the `srcRgn` parameter into the region whose handle you pass in the `dstRgn` parameter; that is, `CopyRgn` makes a duplicate copy of `srcRgn`. When calling `CopyRgn`, pass handles that have been returned by the `NewRgn` function in the `srcRgn` and `dstRgn` parameters.

Once this is done, the region indicated by `srcRgn` may be altered (or even disposed of) without affecting the region indicated by `dstRgn`. The `CopyRgn` procedure does not create the destination region; space must already have been allocated for it by using the `NewRgn` function.

SPECIAL CONSIDERATIONS

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

SetEmptyRgn

To set an existing region to be empty, use the `SetEmptyRgn` procedure.

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

`rgn` A handle to the region to be made empty.

DESCRIPTION

The `SetEmptyRgn` procedure destroys the previous structure of the region whose handle you pass in the `rgn` parameter; it then sets the new structure to the empty region defined by the rectangle (0,0,0,0).

SPECIAL CONSIDERATIONS

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

SetRectRgn

To change the structure of an existing region to that of a rectangle, you can use the `SetRectRgn` procedure.

```
PROCEDURE SetRectRgn (rgn: RgnHandle;
                      left,top,right,bottom: Integer);
```

`rgn` A handle to the region to restructure as a rectangle.

`left` The horizontal coordinate of the upper-left corner of the rectangle to set as the new region.

`top` The vertical coordinate of the upper-left corner of the rectangle to set as the new region.

`right` The horizontal coordinate of the lower-right corner of the rectangle to set as the new region.

`bottom` The vertical coordinate of the lower-right corner of the rectangle to set as the new region.

DESCRIPTION

The `SetRectRgn` procedure destroys the previous structure of the region whose handle you pass in the `rgn` parameter, and it then sets the new structure to the rectangle that you specify in the `left`, `top`, `right`, and `bottom` parameters. If you specify an empty rectangle (that is, `right<=left` or `bottom<=top`), the `SetRectRgn` procedure sets the region to the empty region defined by the rectangle (0,0,0,0).

As an alternative to the `SetRectRgn` procedure, you can change the structure of an existing region to that of a rectangle by using the `RectRgn` procedure, which accepts as a parameter a rectangle instead of four coordinates. The `RectRgn` procedure is described next.

SPECIAL CONSIDERATIONS

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

RectRgn

To change the structure of an existing region to that of a rectangle, you can use the `RectRgn` procedure.

```
PROCEDURE RectRgn (rgn: RgnHandle; r: Rect);
```

`rgn`	A handle to the region to restructure as a rectangle.
`r`	The rectangle structure to use.

DESCRIPTION

The `RectRgn` procedure destroys the previous structure of the `SetRectRgn` procedure, and it then sets the new structure to a rectangle that you specify in the `r` parameter.

As an alternative to the `RectRgn` procedure, you can use the `SetRectRgn` procedure, which accepts as parameters four coordinates instead of a rectangle.

SPECIAL CONSIDERATIONS

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

OffsetRgn

To move a region, use the `OffsetRgn` procedure.

```
PROCEDURE OffsetRgn (rgn: RgnHandle; dh,dv: Integer);
```

`rgn`	A handle to the region to move.
`dh`	The horizontal distance to move the region.
`dv`	The vertical distance to move the region.

DESCRIPTION

The `OffsetRgn` procedure moves the region whose handle you pass in the `rgn` parameter by adding the value you specify in the `dh` parameter to the horizontal coordinates of all points of its region boundary, and by adding the value you specify in the `dv` parameter to the vertical coordinates of all points of its region boundary. If the values of `dh` and `dv` are positive, the movement is to the right and down; if either is negative, the corresponding movement is in the opposite direction. The region retains its size and shape. This doesn't affect the screen unless you subsequently call a routine to draw the region.

The `OffsetRgn` procedure is an especially efficient operation, because most of the data defining a region is stored relative to the `rgnBBox` field in its `Region` record and so isn't actually changed by `OffsetRgn`.

InsetRgn

To shrink or expand a region, use the `InsetRgn` procedure.

```
PROCEDURE InsetRgn (rgn: RgnHandle; dh,dv: Integer);
```

`rgn`	A handle to the region to alter.
`dh`	The horizontal distance to move points on the left and right boundaries in toward or outward from the center.
`dv`	The vertical distance to move points on the top and bottom boundaries in toward or outward from the center.

DESCRIPTION

The `InsetRgn` procedure moves all points on the region boundary of the region whose handle you pass in the `rgn` parameter inward by the vertical distance that you specify in the `dv` parameter and by the horizontal distance that you specify in the `dh` parameter. If you specify negative values for `dh` or `dv`, the `InsetRgn` procedure moves the points outward in that direction.

The `InsetRgn` procedure leaves the region's center at the same position, but moves the outline in (for positive values of `dh` and `dv`) or out (for negative values of `dh` and `dv`). Using `InsetRgn` on a rectangular region has the same effect as using the `InsetRect` procedure.

SPECIAL CONSIDERATIONS

The `InsetRgn` procedure temporarily uses heap space that's twice the size of the original region.

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

SectRgn

To calculate the intersection of two regions, use the `SectRgn` procedure.

```
PROCEDURE SectRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);
```

`srcRgnA` A handle to the first of two regions whose intersection is to be determined.

`srcRgnB` A handle to the second of two regions whose intersection is to be determined.

`dstRgn` A handle to the region to receive the intersection area.

DESCRIPTION

The `SectRgn` procedure calculates the intersection of the two regions whose handles you pass in the `srcRgnA` and `srcRgnB` parameters, and it places the intersection in the region whose handle you pass in the `dstRgn` parameter. If the regions do not intersect, or one of the regions is empty, `SectRgn` sets the destination to the empty region defined by the rectangle (0,0,0,0).

The `SectRgn` procedure does not create a destination region; you must have already allocated memory for it by using the `NewRgn` function.

The destination region may be one of the source regions, if desired.

SPECIAL CONSIDERATIONS

The `SectRgn` procedure may temporarily use heap space that's twice the size of the two input regions.

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

UnionRgn

To calculate the union of two regions, use the `UnionRgn` procedure.

```
PROCEDURE UnionRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);
```

`srcRgnA`	A handle to the first of two regions whose union is to be determined.
`srcRgnB`	A handle to the second of two regions whose union is to be determined.
`dstRgn`	A handle to the region to hold the resulting union area.

DESCRIPTION

The `UnionRgn` procedure calculates the union of the two regions whose handles you pass in the `srcRgnA` and `srcRgnB` parameters, and it places the union in the region whose handle you pass in the `dstRgn` parameter. If both regions are empty, `UnionRgn` sets the destination to the empty region defined by the rectangle (0,0,0,0).

The `UnionRgn` procedure does not create the destination region; you must have already allocated memory for it by using the `NewRgn` function.

The destination region may be one of the source regions, if desired.

SPECIAL CONSIDERATIONS

The `UnionRgn` procedure may temporarily use heap space that's twice the size of the two input regions.

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

DiffRgn

To subtract one region from another, use the `DiffRgn` procedure.

```
PROCEDURE DiffRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);
```

`srcRgnA` A handle to the region to subtract from.

`srcRgnB` A handle to the region to subtract.

`dstRgn` A handle to the region to hold the resulting area.

DESCRIPTION

The `DiffRgn` procedure subtracts the region whose handle you pass in the `srcRgnB` parameter from the region whose handle you pass in the `srcRgnA` parameter and places the difference in the region whose handle you pass in the `dstRgn` parameter. If the first source region is empty, `DiffRgn` sets the destination to the empty region defined by the rectangle (0,0,0,0).

The `DiffRgn` procedure does not create the destination region; you must have already allocated memory for it by using the `NewRgn` function. The destination region may be one of the source regions, if desired.

SPECIAL CONSIDERATIONS

The `DiffRgn` procedure may temporarily use heap space that's twice the size of the two input regions.

XorRgn

To calculate the difference between the union and the intersection of two regions, use the `XorRgn` procedure.

```
PROCEDURE XorRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);
```

`srcRgnA` A handle to the first of two regions to compare.

`srcRgnB` A handle to the second of two regions to compare.

`dstRgn` A handle to the region to hold the result.

DESCRIPTION

The `XorRgn` procedure calculates the difference between the union and the intersection of the regions whose handles you pass in the `srcRgnA` and `srcRgnB` parameters and places the result in the region whose handle you pass in the `dstRgn` parameter.

This does not create the destination region; you must have already allocated memory for it by using the `NewRgn` function.

If the regions are coincident, `XorRgn` sets the destination region to the empty region defined by the rectangle (0,0,0,0).

SPECIAL CONSIDERATIONS

The `XorRgn` procedure may temporarily use heap space that's twice the size of the two input regions.

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

PtInRgn

To determine whether a pixel is within a region, use the `PtInRgn` function.

```
FUNCTION PtInRgn (pt: Point; rgn: RgnHandle): Boolean;
```

`pt`	The point whose pixel is to be checked.
`rgn`	A handle to the region to test.

DESCRIPTION

The `PtInRgn` function checks whether the pixel below and to the right of the point you specify in the `pt` parameter is within the region whose handle you pass in the `rgn` parameter. The `PtInRgn` function returns `TRUE` if so or `FALSE` if not.

RectInRgn

To determine whether a rectangle intersects a region, use the `RectInRgn` function.

```
FUNCTION RectInRgn (r: Rect; rgn: RgnHandle): Boolean;
```

`r`	The rectangle to check for intersection.
`rgn`	A handle to the region to check.

DESCRIPTION

The `RectInRgn` function checks whether the rectangle specified in the `r` parameter intersects the region whose handle you pass in the `rgn` parameter. The `RectInRgn` function returns `TRUE` if the intersection encloses at least 1 bit or `FALSE` if it does not.

SPECIAL CONSIDERATIONS

The `RectInRgn` function sometimes returns `TRUE` when the rectangle merely intersects the region's bounding rectangle. If you need to know exactly whether a given rectangle intersects the actual region, you can use the `RectRgn` procedure (described on page 3-92) to set the rectangle to a region, and call `SectRgn` (described on page 3-94) to see whether the two regions intersect. If the result of `SectRgn` is an empty region, then the rectangle doesn't intersect the region.

EqualRgn

To determine whether two regions have identical sizes, shapes, and locations, use the `EqualRgn` function.

```
FUNCTION EqualRgn (rgnA,rgnB: RgnHandle): Boolean;
```

`srcRgnA`	A handle to the first of two regions to compare.
`srcRgnB`	A handle to the second of two regions to compare.

DESCRIPTION

The `EqualRgn` function compares the two regions whose handles you pass in the `rgnA` and `rgnB` parameters and returns `TRUE` if they're equal or `FALSE` if they're not.

The two regions must have identical sizes, shapes, and locations to be considered equal. Any two empty regions are always equal.

EmptyRgn

To determine whether a region is empty, use the `EmptyRgn` function.

```
FUNCTION EmptyRgn (rgn: RgnHandle): Boolean;
```

`rgn` A handle to the region to test for emptiness.

DESCRIPTION

The `EmptyRgn` function returns `TRUE` if the region whose handle you pass in the `rgn` parameter is an empty region or `FALSE` if it is not.

SEE ALSO

The `EmptyRgn` function does not create an empty region. To create an empty region, you can perform any of the following operations:

- use the `NewRgn` function (described on page 3-87)
- pass the handle to an empty region to the `CopyRgn` procedure (described on page 3-90)
- pass an empty rectangle to either the `SetRectRgn` procedure (described on page 3-91) or the `RectRgn` procedure (described on page 3-92)
- call the `CloseRgn` procedure (described on page 3-89) without a previous call to the `OpenRgn` procedure
- call `CloseRgn` without performing any drawing after calling `OpenRgn`
- pass an empty region to the `OffsetRgn` procedure (described on page 3-93)
- pass an empty region or too large an inset to the `InsetRgn` procedure (described on page 3-93)
- pass two nonintersecting regions to the `SectRgn` procedure (described on page 3-94)
- pass two empty regions to the `UnionRgn` procedure (described on page 3-95)
- pass two identical or nonintersecting regions to the `DiffRgn` (described on page 3-96) or `XorRgn` (described on page 3-96) procedure

Drawing Regions

After defining a region by using the `NewRgn` function and `OpenRgn` procedure, a number of drawing procedures, and the `CloseRgn` procedure, you can draw the region's outline with the `FrameRgn` procedure. You can draw its interior with the `PaintRgn` and `FillRgn` procedures. You can erase it by using the `EraseRgn` procedure, and you can use the `InvertRgn` procedure to reverse the colors of the pixels within it. In all of these procedures, you refer to a region by the handle returned by the `NewRgn` function when you first created the region.

These routines depend on the local coordinate system of the current graphics port. If you draw a region in a graphics port different from the one in which you defined the region, it may not appear in the proper position in the graphics port.

▲ **WARNING**
If any horizontal or vertical line drawn through the region would intersect the region's outline more than 50 times, the results of these graphics operations are undefined. The `FrameRgn` procedure in particular requires that there would be no more than 25 such intersections. ▲

FrameRgn

To draw an outline inside a region, use the `FrameRgn` procedure.

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

`rgn` A handle to the region to frame.

DESCRIPTION

Using the current graphics port's pen pattern, pattern mode, and pen size, the `FrameRgn` procedure draws an outline just inside the region whose handle you pass in the `rgn` parameter. The outline never goes outside the region boundary. The pen location does not change.

If a region is open and being formed, the outside outline of the region being framed is mathematically added to that region's boundary.

SPECIAL CONSIDERATIONS

The `FrameRgn` procedure calls the routines `CopyRgn`, `InsetRgn`, and `DiffRgn`, so `FrameRgn` may temporarily use heap space that's three times the size of the original region.

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

PaintRgn

To paint a region with the graphics pen's pattern and pattern mode, use the `PaintRgn` procedure.

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

`rgn` A handle to the region to paint.

DESCRIPTION

Using the pen pattern and pattern mode for the current graphics port, the `PaintRgn` procedure draws the interior of the region whose handle you pass in the `rgn` parameter. The pen location does not change.

SPECIAL CONSIDERATIONS

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

SEE ALSO

You can use the `FillRgn` procedure, described next, to draw the interior of a region with a pen pattern different from that for the current graphics port.

FillRgn

To fill a region with any available bit pattern, use the `FillRgn` procedure.

```
PROCEDURE FillRgn (rgn: RgnHandle; pat: Pattern);
```

`rgn`	A handle to the region to fill.
`pat`	The bit pattern to use for the fill. Figure 3-3 on page 3-7 illustrates the default fill patterns and the constants you can use to represent them.

DESCRIPTION

Using the `patCopy` pattern mode, the `FillRgn` procedure draws the interior of the region (whose handle you pass in the `rgn` parameter) with the pattern defined in the `Pattern` record that you specify in the `pat` parameter.

This procedure leaves the location of the graphics pen unchanged.

SPECIAL CONSIDERATIONS

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

SEE ALSO

Listing 3-8 on page 3-28 and Listing 3-9 on page 3-29 illustrate how to use this procedure.

You can use the `GetPattern` and `GetIndPattern` routines, described on page 3-126 and page 3-127, respectively, to get a pattern stored in a resource. The `patCopy` pattern mode is described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8. The `Pattern` record is described on page 3-40.

You can use the `PaintRgn` procedure, described in the previous section, to draw the interior of a region with the pen pattern for the current graphics port. To fill a region with a pixel pattern, use the `FillCRegion` procedure, which is described in the chapter "Color QuickDraw."

EraseRgn

To erase a region, use the `EraseRgn` procedure.

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

`rgn`	The region to erase.

DESCRIPTION

Using the `patCopy` pattern mode, the `EraseRgn` procedure draws the interior of the region whose handle you pass in the `rgn` parameter with the background pattern for the current graphics port.

This procedure leaves the location of the graphics pen unchanged.

SPECIAL CONSIDERATIONS

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

SEE ALSO

The `patCopy` pattern mode is described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8.

InvertRgn

To invert the pixels enclosed by a region, use the `InvertRgn` procedure.

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

`rgn` A handle to the region whose pixels are to invert.

DESCRIPTION

The `InvertRgn` procedure inverts the pixels enclosed by the region whose handle you pass in the `rgn` parameter. Every white pixel becomes black and every black pixel becomes white.

This procedure leaves the location of the graphics pen unchanged.

SPECIAL CONSIDERATIONS

The `InvertRgn` procedure was designed for 1-bit images in basic graphics ports. This procedure operates on color pixels in color graphics ports, but the results are predictable only with 1-bit or direct pixels. For indexed pixels, Color QuickDraw performs the inversion on the pixel indexes, which means the results depend entirely on the contents of the CLUT (which is described in the chapter "Color QuickDraw"). The eight colors used in basic QuickDraw are stored in a color table represented by the global variable `QDColors`. To display those eight basic QuickDraw colors on an indexed device, Color QuickDraw uses the Color Manager to obtain indexes to the colors in the CLUT that best map to the colors in the `QDColors` color table. Because the index, not the color value, is inverted, the results are unpredictable.

Inversion works better for direct pixels. Inverting a pure green, for example, that has red, green, and blue component values of $0000, $FFFF, and $0000 results in magenta, which has component values of $FFFF, $0000, and $FFFF.

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

Scaling and Mapping Points, Rectangles, Polygons, and Regions

QuickDraw provides procedures to help you map points, rectangles, regions, and polygons from one rectangle to another. You can scale rectangles, regions, and polygons into other rectangles.

To derive vertical and horizontal scaling factors from the proportions of two rectangles, use the `ScalePt` procedure. To map a point in one rectangle to an equivalent position in another rectangle, use the `MapPt` procedure. To map and scale a rectangle within one rectangle to another rectangle, use the `MapRect` procedure. To map and scale a region within one rectangle to another rectangle, use the `MapRgn` procedure. To map and scale a polygon within one rectangle to another rectangle, use the `MapPoly` procedure.

If the points or rectangles supplied to these routines are defined in a graphics port other than your current graphics port, you must convert them to the local coordinate system of your current graphics port. You can accomplish this by using the `SetPort` procedure to change to the graphics port containing the points or rectangles, using the `LocalGlobal` procedure to convert their locations to global coordinates, using `SetPort` to return to your starting graphics port, and then using the `GlobalToLocal` procedure to convert the locations of points or rectangles to the local coordinates of your current graphics port. These procedures are described in the chapter "Basic QuickDraw."

ScalePt

To scale a height and width according to the proportions of two rectangles, use the `ScalePt` procedure.

```
PROCEDURE ScalePt (VAR pt: Point; srcRect,dstRect: Rect);
```

`pt`	On input, an initial height and width (specified in the two fields of a `Point` record) to scale; upon completion, vertical and horizontal scaling factors derived by multiplying the height and width by ratios of the height and width of the rectangle in the `srcRect` parameter to the height and width of the rectangle in the `dstRect` parameter.
`srcRect`	A rectangle. The ratio of this rectangle's height to the height of the rectangle in the `dstRect` parameter provides the vertical scaling factor, and the ratio of this rectangle's width to the width of the rectangle in the `dstRect` parameter provides the horizontal scaling factor.
`dstRect`	A rectangle compared to the rectangle in the `srcRect` parameter to determine vertical and horizontal scaling factors.

DESCRIPTION

The `ScalePt` procedure produces horizontal and vertical scaling factors from the proportions of two rectangles. You can use `ScalePt`, for example, to scale the dimensions of the graphics pen.

You specify an initial height and width to scale in the `pt` parameter. This parameter is of type `Point`, although you don't pass coordinates in this parameter. Instead, you pass an initial height to scale in the `v` (or vertical) field of the `Point` record, and you pass an initial width to scale in the `h` (or horizontal) field.

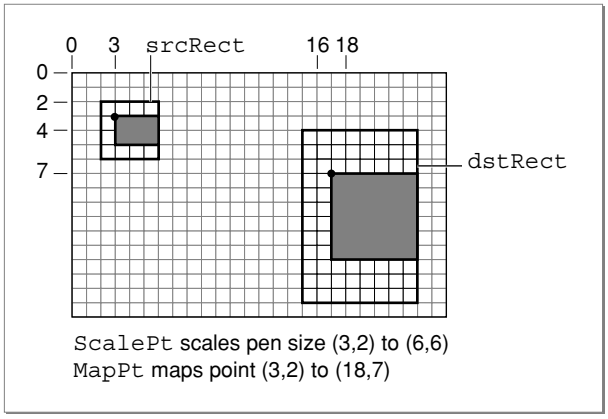
The `ScalePt` procedure scales these measurements by multiplying the initial height you specify in the `pt` parameter by the ratio of the height of the rectangle you specify in the `dstRect` parameter to the height of the rectangle you specify in the `srcRect` parameter, and by multiplying the initial width in the `pt` parameter by the ratio of the width of the `dstRect` rectangle to the width of the `srcRect` rectangle. The `ScalePt` procedure returns the result in the `pt` parameter.

In Figure 3-23, where the width of the `dstRect` rectangle is twice the width of the `srcRect` rectangle, and its height is three times the height of `srcRect`, `ScalePt` scales the width of the graphics pen from 3 to 6 and scales its height from 2 to 6.

SPECIAL CONSIDERATIONS

The minimum value `ScalePt` returns is (1,1).

Figure 3-23 Using `ScalePt` and `MapPt`



MapPt

To map a point in one rectangle to an equivalent position in another rectangle, use the `MapPt` procedure.

```
PROCEDURE MapPt (VAR pt: Point; srcRect,dstRect: Rect);
```

`pt`	Upon input, the point in the source rectangle to map; upon completion, its mapped position in the destination rectangle.
`srcRect`	The source rectangle containing the original point.
`dstRect`	The destination rectangle in which the point will be mapped.

DESCRIPTION

The `MapPt` procedure maps a point in one rectangle to an equivalent position in another rectangle.

In the `pt` parameter, you specify a point that lies within the rectangle that you specify in the `srcRect` parameter. The `MapPt` procedure maps this point to a similarly located point within the rectangle that you specify in the `dstRect` parameter—that is, to where it would fall if it were part of a drawing being expanded or shrunk to fit the destination rectangle. The `MapPt` procedure returns the location of the mapped point in the `pt` parameter. For example, a corner point of the source rectangle would be mapped to the corresponding corner point of the destination rectangle in `dstRect`, and the center of the source rectangle would be mapped to the center of destination rectangle.

The source and destination rectangles may overlap, and the point you specify need not actually lie within the source rectangle.

In Figure 3-23 on page 3-105, the point (3,2) in the source rectangle is mapped to (18,7) in the destination rectangle.

SEE ALSO

If you're going to draw inside the destination rectangle, you'll probably also want to scale the graphics pen size accordingly with the `ScalePt` procedure, described in the previous section.

MapRect

To map and scale a rectangle within one rectangle to another rectangle, use the `MapRect` procedure.

```
PROCEDURE MapRect (VAR r: Rect; srcRect,dstRect: Rect);
```

`r`	Upon input, the rectangle to map; upon completion, the mapped rectangle.
`srcRect`	The rectangle containing the rectangle to map.
`dstRect`	The rectangle in which the new rectangle will be mapped.

DESCRIPTION

The `MapRect` procedure takes a rectangle within one rectangle and maps and scales it to another rectangle. In the `r` parameter, you specify a rectangle that lies within the rectangle that you specify in the `srcRect` parameter. By calling the `MapPt` procedure to map the upper-left and lower-right corners of the rectangle in the `r` parameter, `MapRect` maps and scales it to the rectangle that you specify in the `dstRect` parameter. The `MapRect` procedure returns the newly mapped rectangle in the `r` parameter.

MapRgn

To map and scale a region within one rectangle to another rectangle, use the `MapRgn` procedure.

```
PROCEDURE MapRgn (rgn: RgnHandle; srcRect,dstRect: Rect);
```

`rgn`	A handle to a region. Upon input, this is the region to map. Upon completion, this region is the one mapped to a new location.
`srcRect`	The rectangle containing the region to map.
`dstRect`	The rectangle in which the new region will be mapped.

DESCRIPTION

The `MapRgn` procedure takes a region within one rectangle and maps and scales it to another rectangle. In the `rgn` parameter, you specify a handle to a region that lies within the rectangle that you specify in the `srcRect` parameter. By calling the `MapPt` procedure to map all the points of the region in the `rgn` parameter, `MapRgn` maps and scales it to the rectangle that you specify in the `dstRect` parameter. The `MapRgn` procedure returns the result in the region whose handle you initially passed in the `rgn` parameter.

The `MapRgn` procedure is useful for determining whether a region operation will exceed available memory. By mapping a large region into a smaller one and performing the operation (without actually drawing), you can estimate how much memory will be required by the anticipated operation.

SPECIAL CONSIDERATIONS

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

MapPoly

To map and scale a polygon within one rectangle to another rectangle, use the `MapPoly` procedure.

```
PROCEDURE MapPoly (poly: PolyHandle; srcRect,dstRect: Rect);
```

`poly`	A handle to a polygon. Upon input, this is the polygon to map. Upon completion, this polygon is the one mapped to a new location.
`srcRect`	The rectangle containing the polygon.
`dstRect`	The rectangle in which the new region will be mapped.

DESCRIPTION

The `MapPoly` procedure takes a polygon within one rectangle and maps and scales it to another rectangle. In the `poly` parameter, you specify a handle to a polygon that lies within the rectangle that you specify in the `srcRect` parameter. By calling the `MapPt` procedure to map all the points that define the polygon specified in the `poly` parameter, `MapPoly` maps and scales it to the rectangle that you specify in the `dstRect` parameter. The `MapPoly` procedure returns the result in the polygon whose handle you initially passed in the `poly` parameter.

Similar to the `MapRgn` procedure described in the previous section, the `MapPoly` procedure is useful for determining whether a polygon operation will exceed available memory.

Calculating Black-and-White Fills

QuickDraw provides the `SeedFill` and `CalcMask` procedures to help you determine the results of filling operations on portions of bitmaps. (Procedures for determining filling operations on pixel maps—namely, `SeedCFill` and `CalcCMask`—are described in the chapter "Color QuickDraw.")

The `SeedFill` procedure produces a mask showing where bits would be filled from a starting point, like the paint pouring from the MacPaint® paint-bucket tool. The `CalcMask` procedure produces a mask showing where paint could *not* flow from any of the outer edges of a rectangle. You can use the resulting masks to transfer portions of bit images from one graphics port to another with the `CopyBits` or `CopyMask` procedure, both of which are described in "Copying Images" beginning on page 3-112.

SeedFill

To determine how far filling will extend from a seeding point, use the `SeedFill` procedure.

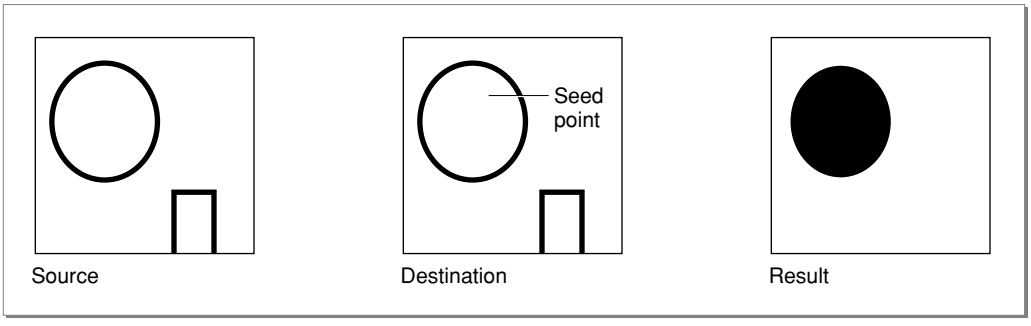
```
PROCEDURE SeedFill (srcPtr,dstPtr: Ptr;
                    srcRow,dstRow,height,words,
                    seedH,seedV: Integer);
```

`srcPtr`	A pointer to the source bit image.
`dstPtr`	On input, a pointer to the destination bit image; upon return, a pointer to the bitmap containing the resulting mask.
`srcRow`	Row width of the source bitmap.
`dstRow`	Row width of the destination bitmap.
`height`	Height (in pixels) of the fill rectangle.
`words`	Width (in words) of the fill rectangle.
`seedH`	The horizontal offset (in pixels) at which to begin filling the destination bit image.
`seedV`	The vertical offset (in pixels) at which to begin filling the destination bit image.

DESCRIPTION

The `SeedFill` procedure produces a mask showing where bits in an image can be filled from a starting point, like the paint pouring from the MacPaint paint-bucket tool. The `SeedFill` returns this mask in the `dstPtr` parameter. This mask is a bitmap filled with 1's only where the pixels in the source image can be filled. This is illustrated in Figure 3-24. You can then use this mask with the `CopyBits`, `CopyMask`, and `CopyDeepMask` procedures.

Figure 3-24 A source image and its resulting mask produced by the `SeedFill` procedure

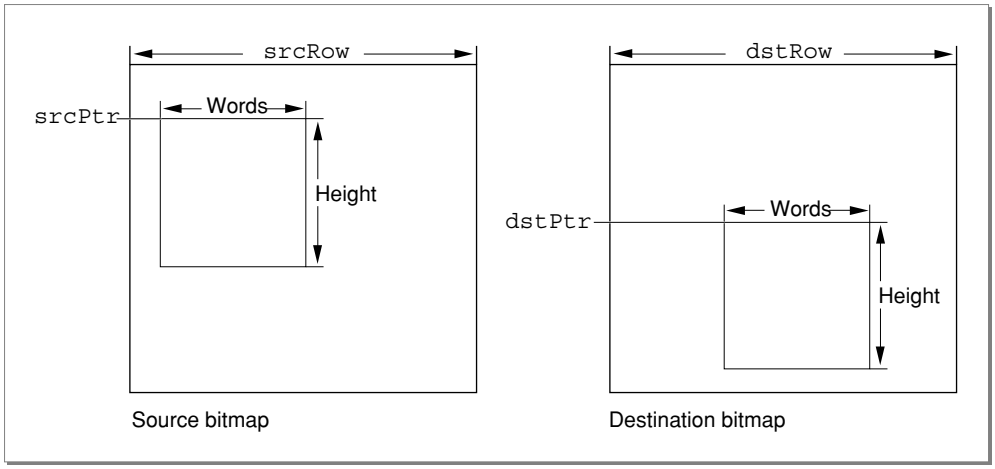


Point to the bit image you want to fill with the `srcPtr` parameter, which can point to the image's base address or a word boundary within the image. Specify a pixel height and word width with the `height` and `words` parameters to define a fill rectangle that delimits the area you want to fill. The fill rectangle can be the entire bit image or a subset of it. Point to a destination image with the `dstPtr` parameter. Specify the row widths of the source and destination bitmaps (their `rowBytes` values) with the `srcRow` and `dstRow` parameters. (The bitmaps can be different sizes, but they must be large enough to contain the fill rectangle at the origins specified by the `srcPtr` and `dstPtr` parameters.) Figure 3-25 illustrates these parameters for the source and destination bit images.

You specify where to begin filling with the `seedH` and `seedV` parameters: they specify a horizontal and vertical offset in pixels from the origin of the image pointed to by the `srcPtr` parameter. The `SeedFill` procedure calculates contiguous pixels from that point out to the boundaries of the fill rectangle, and it stores the result in the bit image pointed to by the `dstPtr` parameter.

Calls to `SeedFill` are not clipped to the current port and are not stored into QuickDraw pictures.

Figure 3-25 Parameters for the `SeedFill` and `CalcMask` procedures



SEE ALSO

For color graphics ports, use the `SeedCFill` procedure, which is described in the chapter "Color QuickDraw."

CalcMask

To determine where filling will not occur when filling from the outside of a rectangle, use the `CalcMask` procedure.

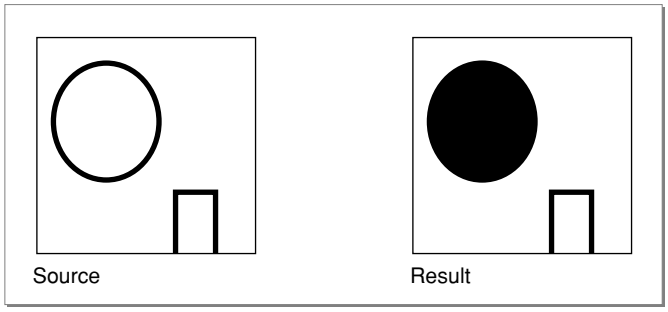
```
PROCEDURE CalcMask (srcPtr,dstPtr: Ptr;
                    srcRow,dstRow,height,words: Integer);
```

`srcPtr`	A pointer to the source bit image.
`dstPtr`	A pointer to the destination bit image.
`srcRow`	Row width of the source bitmap.
`dstRow`	Row width of the destination bitmap.
`height`	Height (in pixels) of the fill rectangle.
`words`	Width (in words) of the fill rectangle.

DESCRIPTION

The `CalcMask` procedure produces a bit image with 1's in all pixels to which paint could *not* flow from any of the outer edges of the rectangle. You can use this bit image as a mask with the `CopyBits` or `CopyMask` procedure. As illustrated in Figure 3-26, a hollow object produces a solid mask, but an open object produces a mask of itself.

Figure 3-26 A source image and the resulting mask produced by the `CalcMask` procedure



As with the `SeedFill` procedure, point to the bit image you want to fill with the `srcPtr` parameter, which can point to the image's base address or a word boundary within the image. Specify a pixel height and word width with the `height` and `words` parameters to define a fill rectangle that delimits the area you want to fill. The fill rectangle can be the entire bit image or a subset of it. Point to a destination image with the `dstPtr` parameter. Specify the row widths of the source and destination bitmaps (their `rowBytes` values) with the `srcRow` and `dstRow` parameters. (The bitmaps can be different sizes, but they must be large enough to contain the fill rectangle at the origins specified by `srcPtr` and `dstPtr`.)

Figure 3-25 on page 3-110 illustrates the parameters for the source and destination bit images.

Calls to `CalcMask` are not clipped to the current port and are not stored into QuickDraw pictures.

SEE ALSO

For color graphics ports, use the `CalcCMask` procedure, which is described in the chapter "Color QuickDraw."

Copying Images

QuickDraw provides three procedures for copying portions of bitmaps from one graphics port or offscreen graphics world into another graphics port. The `CopyBits` procedure allows you to copy using source modes and to clip and resize during the copy operation. The `CopyMask` procedure allows you to mask areas where you want the copy operation to occur. These procedures also allow you to use pixel maps instead of bitmaps when your application runs on systems supporting Color QuickDraw.

The `CopyDeepMask` procedure, which is available to basic QuickDraw only in System 7, combines the functionality of both `CopyBits` and `CopyMask`.

CopyBits

You can use the `CopyBits` procedure to copy a portion of a bitmap or a pixel map from one graphics port (or offscreen graphics world) into another graphics port.

```
PROCEDURE CopyBits (srcBits,dstBits: BitMap;
                    srcRect,dstRect: Rect; mode: Integer;
                    maskRgn: RgnHandle);
```

`srcBits`	The source `BitMap` record.
`dstBits`	The destination `BitMap` record.
`srcRect`	The source rectangle.
`dstRect`	The destination rectangle.
`mode`	One of the eight source modes in which the copy is to be performed.
`maskRgn`	A region to use as a clipping mask.

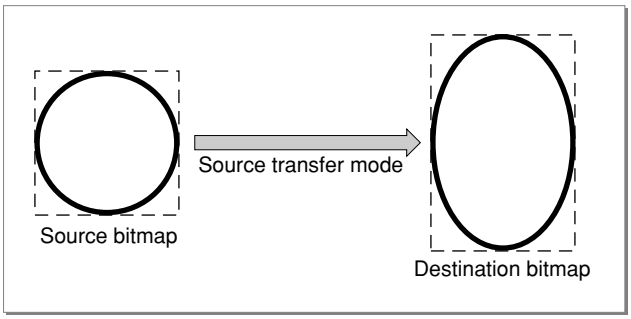
DESCRIPTION

The `CopyBits` procedure transfers any portion of a bitmap between two basic graphics ports, or any portion of a pixel map between two color graphics ports. You can use `CopyBits` to move offscreen graphic images into an onscreen window, to blend colors for the image in a pixel map, and to shrink and expand images.

Specify a source bitmap in the `srcBits` parameter and a destination bitmap in the `dstBits` parameter. When copying images between color graphics ports, you must coerce each `CGrafPort` record to a `GrafPort` record, dereference the `portBits` fields of each, and then pass these "bitmaps" in the `srcBits` and `dstBits` parameters. If your application copies a pixel image from a color graphics port called `MyColorPort`, for example, you could specify `GrafPtr(MyColorPort)^.portBits` in the `srcBits` parameter. In a `CGrafPort` record, the high 2 bits of the `portVersion` field are set. This field, which shares the same position in a `CGrafPort` record as the `portBits.rowBytes` field in a `GrafPort` record, indicates to `CopyBits` that you have passed it a handle to a pixel map rather than a bitmap.

Using the `srcRect` and `dstRect` parameters, you can specify identically or differently sized source and destination rectangles; for differently sized rectangles, `CopyBits` scales the source image to fit the destination. As shown in Figure 3-27, for example, if the bit image is a circle in a square source rectangle, and the destination rectangle is not square, the bit image appears as an oval in the destination. When you specify rectangles in the `srcRect` and `dstRect` parameters, use the local coordinate systems of, respectively, the source and destination graphics ports.

Figure 3-27 Using `CopyBits` to stretch an image



In the `mode` parameter, specify one of the following source modes for transferring the bits from a source bitmap to a destination bitmap:

```
CONST              {source modes for basic graphics ports}
   srcCopy   = 0; {where source pixel is black, force }
                  { destination pixel black; where source pixel }
                  { is white, force destination pixel white}
   srcOr     = 1; {where source pixel is black, force }
                  { destination pixel black; where source pixel }
                  { is white, leave destination pixel unaltered}
   srcXor    = 2; {where source pixel is black, invert }
                  { destination pixel; where source pixel is }
                  { white, leave destination pixel unaltered}
   srcBic    = 3; {where source pixel is black, force }
                  { destination pixel white; where source pixel }
                  { is white, leave destination pixel unaltered}
   notSrcCopy
             = 4; {where source pixel is black, force }
                  { destination pixel white; where source pixel }
                  { is white, force destination pixel black}
   notSrcOr  = 5; {where source pixel is black, leave }
                  { destination pixel unaltered; where source }
                  { pixel is white, force destination pixel black}
   notSrcXor = 6; {where source pixel is black, leave }
                  { destination pixel unaltered; where source }
                  { pixel is white, invert destination pixel}
   notSrcBic = 7; {where source pixel is black, leave }
                  { destination pixel unaltered; where source }
                  { pixel is white, force destination pixel white}
```

On computers running System 7, you can add dithering to any source mode by adding the following constant or the value it represents to the source mode:

```
CONST ditherCopy  = 64; {add to source mode for dithering}
```

Dithering is a technique that mixes existing colors to create the effect of additional colors. It also improves images that you shrink or that you copy from a direct pixel device to an indexed device. The `CopyBits` procedure always dithers images when shrinking them between pixel maps on direct devices.

To use highlighting, you can add this constant or its value to the source mode:

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

With highlighting, QuickDraw replaces the background color with the highlight color when your application copies images between graphics ports. This has the visual effect of using a highlighting pen to select the object. (The global variable `HiliteRGB` is read from parameter RAM when the machine starts. Basic graphics ports use the color stored in the `HiliteRGB` global variable as the highlight color. Color graphics ports default to the `HiliteRGB` global variable, but can be overridden by the `HiliteColor` procedure, described in the chapter "Color QuickDraw.")

When transferring pixels from a source pixel map to a destination pixel map, Color QuickDraw interprets the source mode constants differently than basic QuickDraw does. These constants have the following effects under Color QuickDraw:

```
CONST               {source modes for color graphics ports}
   srcCopy    = 0;{determine how close the color of the source }
                  { pixel is to black, and assign this relative }
                  { amount of foreground color to the }
                  { destination pixel; determine how close the }
                  { color of the source pixel is to white, and }
                  { assign this relative amount of background }
                  { color to the destination pixel}
   srcOr      = 1;{determine how close the color of the source }
                  { pixel is to black, and assign this relative }
                  { amount of foreground color to the }
                  { destination pixel}
   srcXor     = 2;{where source pixel is black, invert the }
                  { destination pixel--for a colored destination }
                  { pixel, use the complement of its color }
                  { if the pixel is direct, invert its index if }
                  { the pixel is indexed}
   srcBic     = 3;{determine how close the color of the source }
                  { pixel is to black, and assign this relative }
                  { amount of background color to the }
                  { destination pixel}
   notSrcCopy = 4;{determine how close the color of the source }
                  { pixel is to black, and assign this relative }
                  { amount of background color to the }
                  { destination pixel; determine how close the }
                  { color of the source pixel is to white, and }
                  { assign this relative amount of foreground }
                  { color to the destination pixel}
   notSrcOr  = 5; {determine how close the color of the source }
                  { pixel is to white, and assign this relative }
                  { amount of foreground color to the }
                  { destination pixel}
```

```
   notSrcXor = 6; {where source pixel is white, invert the }
                  { destination pixel--for a colored destination }
                  { pixel, use the complement of its color }
                  { if the pixel is direct, invert its index if }
                  { the pixel is indexed}
   notSrcBic = 7; {determine how close the color of the source }
                  { pixel is to white, and assign this relative }
                  { amount of background color to the }
                  { destination pixel}
```

When you use `CopyBits` on a computer running Color QuickDraw, you can also specify one of the following transfer modes in the `mode` parameter:

```
CONST {arithmetic transfer modes available in Color QuickDraw}
   blend       = 32; {replace destination pixel with a blend }
                     { of the source and destination pixel }
                     { colors; if the destination is a bitmap or }
                     { 1-bit pixel map, revert to srcCopy mode}
   addPin      = 33; {replace destination pixel with the sum of }
                     { the source and destination pixel colors-- }
                     { up to a maximum allowable value; if }
                     { the destination is a bitmap or }
                     { 1-bit pixel map, revert to srcBic mode}
   addOver     = 34; {replace destination pixel with the sum of }
                     { the source and destination pixel colors-- }
                     { but if the value of the red, green, or }
                     { blue component exceeds 65,536, then }
                     { subtract 65,536 from that value; if the }
                     { destination is a bitmap or 1-bit }
                     { pixel map, revert to srcXor mode}
   subPin      = 35; {replace destination pixel with the }
                     { difference of the source and destination }
                     { pixel colors--but not less than a minimum }
                     { allowable value; if the destination }
                     { is a bitmap or 1-bit pixel map, revert to }
                     { srcOr mode}
   transparent = 36; {replace the destination pixel with the }
                     { source pixel if the source pixel isn't }
                     { equal to the background color}
```

```
addMax      = 37; {compare the source and destination pixels, }
                  { and replace the destination pixel with }
                  { the color containing the greater }
                  { saturation of each of the RGB components; }
                  { if the destination is a bitmap or }
                  { 1-bit pixel map, revert to srcBic mode}
subOver     = 38; {replace destination pixel with the }
                  { difference of the source and destination }
                  { pixel colors--but if the value of the }
                  { red, green, or blue component is }
                  { less than 0, add the negative result to }
                  { 65,536; if the destination is a bitmap or }
                  { 1-bit pixel map, revert to srcXor mode}
adMin       = 39; {compare the source and destination pixels, }
                  { and replace the destination pixel with }
                  { the color containing the lesser }
                  { saturation of each of the RGB components; }
                  { if the destination is a bitmap or }
                  { 1-bit pixel map, revert to srcOr mode}
```

You can pass a region handle in the `MaskRgn` parameter to specify a mask region; the resulting image is always clipped to this mask region and to the boundary rectangle of the destination bitmap. If the destination bitmap is the current graphics port's bitmap, it's also clipped to the intersection of the graphics port's clipping region and visible region. If you don't want to clip to a masking region, just pass `NIL` for the `maskRgn` parameter.

SPECIAL CONSIDERATIONS

When you use the `CopyBits` procedure to transfer an image between pixel maps, the source and destination images may be of different pixel depths, of different sizes, and they may have different color tables. However, `CopyBits` assumes that the destination pixel map uses the same color table as the color table for the current `GDevice` record. (This is because the Color Manager requires an inverse table for translating the color table from the source pixel map to the destination pixel map.)

The `CopyBits` procedure applies the foreground and background colors of the current graphics port to the image in the destination pixel map (or bitmap), even if the source image is a bitmap. This causes the foreground color to replace all black pixels in the destination and the background color to replace all white pixels. To avoid unwanted coloring of the image, use the `RGBForeColor` procedure to set the foreground to black and use the `RGBBackColor` procedure to set the background to white before calling `CopyBits`.

The source bitmap or pixel map must not occupy more memory than half the available stack space. The stack space required by `CopyBits` is roughly five times the value of the `rowBytes` field of the source pixel map: one `rowBytes` value for the pixel map (or bitmap), an additional `rowBytes` value for dithering, another `rowBytes` value when stretching or shrinking the source pixel map into the destination, another `rowBytes` value for any color map changing, and a fifth additional `rowBytes` value for any color aliasing. If there is insufficient memory to complete a `CopyBits` operation in Color QuickDraw, the `QDError` function (described in the chapter "Color QuickDraw" in this book) returns the result code –143.

If you use `CopyBits` to copy between two graphics ports that overlap, you must first use the `LocalToGlobal` procedure to convert to global coordinates, and then specify the global variable `screenBits` for both the `srcBits` and `dstBits` parameters.

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

If you are reading directly from a NuBus™ video card with a base address of Fs00000 and there is not a card in the slot (s–1) below it, `CopyBits` reads addresses less than the base address of the pixel map. This causes a bus error. To work around the problem, remap the `baseAddr` field of the pixel map in your video card to at least 20 bytes above the NuBus boundary; an address link of Fs000020 precludes the problem.

SEE ALSO

Listing 3-11 on page 3-33 illustrates how to use `CopyBits` to scale an image when copying it from one window into another. Source modes are described in "Boolean Transfer Modes With 1-Bit Pixels" beginning on page 3-8. Plate 2 at the front of this book illustrates how to use `CopyBits` to colorize an image in a color graphics port; Listing 4-5 on page 4-35 in the chapter "Color QuickDraw" shows the sample code that produced this plate. Listing 6-1 on page 6-5 in the chapter "Offscreen Graphics Worlds" illustrates how to use `CopyBits` to copy an image from an offscreen graphics world to an onscreen color graphics port.

Dithering, pixel maps, color graphics ports, the `RGBForeColor` and `RGBBackColor` procedures, and color tables are explained in the chapter "Color QuickDraw." The `LocalToGlobal` procedure is described in the chapter "Basic QuickDraw." The `GDevice` record is described in the chapter "Graphics Devices." Inverse tables and the Color Manager are described in the chapter "Color Manager" in *Inside Macintosh: Advanced Color Imaging*.

"Copying Pixels Between Color Graphics Ports" in the chapter "Color QuickDraw" describes in greater detail how to use `CopyBits` to transfer colored images.

The `CopyDeepMask` procedure (described on page 3-120) combines the functions of the `CopyBits` and `CopyMask` procedures. (The `CopyMask` procedure is described next.)

CopyMask

You can use the `CopyMask` procedure to copy a bit or pixel image from one graphics port (or offscreen graphics world) into another graphics port only where the bits in a mask are set to 1.

```
PROCEDURE CopyMask (srcBits,maskBits,dstBits: BitMap;
                    srcRect,maskRect,dstRect: Rect);
```

`srcBits`	The source `BitMap` record.
`maskBits`	The mask `BitMap` record.
`dstBits`	The destination `BitMap` record.
`srcRect`	The source rectangle.
`maskRect`	The mask rectangle. This must be the same size as the rectangle passed in the `srcRect` parameter.
`dstRect`	The destination rectangle.

DESCRIPTION

The `CopyMask` procedure copies the source bitmap or pixel map that you specify in the `srcBits` parameter to a destination bitmap or pixel map that you specify in the `dstBits` parameter—but only where the bits of the mask bitmap or pixel map that you specify in the `maskBits` parameter are set to 1. When copying images between color graphics ports, you must coerce each `CGrafPort` record to a `GrafPort` record, dereference the `portBits` fields of each, and then pass these "bitmaps" in the `srcBits` and `dstBits` parameters. If your application copies a pixel image from a color graphics port called `MyColorPort`, for example, you could specify `GrafPtr(MyColorPort)^.portBits` in the `srcBits` parameter.

Using the `srcRect` and `dstRect` parameters, you can specify identically or differently sized source and destination rectangles; for differently sized rectangles, `CopyMask` scales the source image to fit the destination. When you specify rectangles in the `srcRect` and `dstRect` parameters, use the local coordinate systems of, respectively, the source and destination graphics ports.

The rectangle you pass in the `maskRect` parameter selects the portion of the bitmap or pixel map that you specify in the `maskBits` parameter to use as the mask.

If you specify pixel maps to `CopyMask`, they may range from 1 to 32 pixels in depth. The pixel depth of the mask that you specify in the `maskBits` parameter is applied as a filter between the source and destination pixel maps that you specify in the `srcBits` and `dstBits` parameters. A black mask pixel value means that the copy operation is to take the source pixel; a white value means that the copy operation is to take the destination

pixel. Intermediate values specify a weighted average, which is calculated on a color component basis. For each pixel's color component value, the calculation is

(1 – mask) × source + (mask) × destination

Thus high mask values for a pixel's color component reduce that component's contribution from the source `PixMap` record.

SPECIAL CONSIDERATIONS

Calls to `CopyMask` are not recorded in pictures and do not print.

See the list of special considerations for the `CopyBits` procedure beginning on page 3-117; these considerations also apply to `CopyMask`.

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

SEE ALSO

You can use the bitmap returned by the `CalcMask` procedure, described on page 3-111, as the mask in order to implement a mask copy similar to that performed by the MacPaint lasso tool. In the same way, you could use the pixel map returned by the `CalcCMask` procedure, described in the chapter "Color QuickDraw."

The chapter "Color QuickDraw" describes in more detail how to use `CopyMask` in a Color QuickDraw environment. Plate 3 at the front of this book illustrates how to use different colors in the mask to produce different effects in the destination pixel map; Listing 6-2 on page 6-10 in the chapter "Offscreen Graphics Worlds" shows the code that produced this plate. Plate 4 at the front of this book provides another illustration of the effects of the source and mask pixel maps on the destination pixel map.

The `CopyDeepMask` procedure (described next) combines the functions of the `CopyMask` and `CopyBits` procedures.

CopyDeepMask

To use a mask when copying bitmaps or pixel maps between graphics ports (or from an offscreen graphics world into a graphics port), you can use the `CopyDeepMask` procedure, which combines the effects of the `CopyBits` and `CopyMask` procedures.

```
PROCEDURE CopyDeepMask  (srcBits: BitMap; maskBits: BitMap;
                         dstBits: BitMap; srcRect: Rect;
                         maskRect: Rect; dstRect: Rect;
                         mode: Integer; maskRgn: RgnHandle);
```

`srcBits`	The source `BitMap` record.
`maskBits`	The masking `BitMap` record.
`dstBits`	The destination `BitMap` record.
`srcRect`	The source rectangle.
`maskRect`	The mask rectangle. This must be the same size as the rectangle passed in the `srcRect` parameter.
`dstRect`	The destination rectangle.
`mode`	The source mode.
`maskRgn`	The mask clipping region.

DESCRIPTION

The `CopyDeepMask` procedure transfers a bitmap between two basic graphics ports or a pixel map between two color graphics ports. You specify a mask to `CopyDeepMask` so that it transfers the source image to the destination image only where the bits of the mask are set to 1. You can use `CopyDeepMask` to move offscreen graphic images into an onscreen window, to blend colors for the image in a pixel map, and to shrink and expand images.

Specify a source bitmap in the `srcBits` parameter and a destination bitmap in the `dstBits` parameter. Specify a mask in the `maskBits` parameter. When copying images between color graphics ports, you must coerce each port's `CGrafPort` record to a `GrafPort` record, dereference the `portBits` fields of each, and then pass these "bitmaps" in the `srcBits` and `dstBits` parameters. If your application copies a pixel image from a color graphics port called `MyColorPort`, for example, you could specify `GrafPtr(MyColorPort)^.portBits` in the `srcBits` parameter. The transfer can be performed in any of the transfer modes—with or without adding the `ditherCopy` constant—that are available to the `CopyBits` procedure, described beginning on page 3-112.

Using the `srcRect` and `dstRect` parameters, you can specify identically or differently sized source and destination rectangles; for differently sized rectangles, `CopyDeepMask` scales the source image to fit the destination. When you specify rectangles in the `srcRect` and `dstRect` parameters, use the local coordinate systems of, respectively, the source and destination graphics ports.

The result (in the parameter `dstBits`) is clipped to the mask region that you specify in the `maskRgn` parameter, and to the boundary rectangle that you specify in the `dstRect` parameter. The rectangle you pass in the `maskRect` parameter selects the portion of the bitmap or pixel map that you specify in the `maskBits` parameter to use as the mask. If you don't want to clip to the mask region, specify `NIL` in the `maskRgn` parameter.

If you specify pixel maps to `CopyDeepMask`, they may range from 1 to 32 pixels in depth. The pixel depth of the mask that you specify in the `maskBits` parameter is applied as a filter between the source and destination pixel maps that you specify in the `srcBits` and `dstBits` parameters. A black mask pixel value means that the copy operation is to take the source pixel; a white value means that the copy operation is to take the destination pixel. Intermediate values specify a weighted average, which is calculated on a color component basis. For each pixel's color component value, the calculation is

(1 – mask) × source + (mask) × destination

Thus high mask values for a pixel's color component reduce that component's contribution from the source `PixMap` record.

SPECIAL CONSIDERATIONS

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

As with the `CopyMask` procedure, calls to `CopyDeepMask` are not recorded in pictures and do not print.

See the list of special considerations for the `CopyBits` procedure beginning on page 3-117; these considerations also apply to `CopyDeepMask`.

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

SEE ALSO

The chapter "Color QuickDraw" describes in more detail how to use `CopyDeepMask` in a Color QuickDraw environment.

Drawing With the Eight-Color System

On a color screen, you can draw using eight predefined colors, even when you are using a basic graphics port. Although basic QuickDraw graphics routines were designed for black-and-white drawing, they also include rudimentary color capabilities. Because Color QuickDraw also supports this system, it is compatible across all Macintosh platforms. (This section describes the rudimentary color routines included in basic QuickDraw. See the next chapter, "Color QuickDraw," for information about more sophisticated color use in your application.)

A pair of fields in a `GrafPort` record, `fgColor` and `bkColor`, specify a foreground and background color. The foreground color is the color of the "ink" used to frame, fill, and paint. By default, the foreground color is black. The background color is the color of the pixels in the bitmap wherever no drawing has taken place. By default, the background color is white. However, you can use the `ForeColor` and `BackColor` procedures to change these fields. When printing, however, use the `ColorBit` procedure to set the foreground color.

In System 7, these Color QuickDraw routines are available to basic QuickDraw: `RGBForeColor`, `RGBBackColor`, `GetForeColor`, and `GetBackColor`. Described in the next chapter, "Color QuickDraw," these routines can also assist you in manipulating the eight-color system of basic QuickDraw.

ForeColor

To change the color of the "ink" used for framing, painting, and filling on computers that support only basic QuickDraw, you can use the `ForeColor` procedure.

```
PROCEDURE ForeColor (color: LongInt);
```

`color` One of eight color values. You can use the following constants to represent these values:

```
CONST
   whiteColor     = 30;
   blackColor     = 33;
   yellowColor    = 69;
   magentaColor   = 137;
   redColor       = 205;
   cyanColor      = 273;
   greenColor     = 341;
   blueColor      = 409;
```

DESCRIPTION

The `ForeColor` procedure sets the foreground color for the current graphics port to the color that you specify in the `color` parameter. When you draw with the `patCopy` and `srcCopy` transfer modes, for example, black pixels are drawn in the color you specify with `ForeColor`.

SPECIAL CONSIDERATIONS

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

SEE ALSO

All nonwhite colors appear as black on black-and-white screens. Before you use `ForeColor`, you can use the `DeviceLoop` procedure, which is described in the chapter "Graphics Devices," to determine the color characteristics of the current screen.

In System 7, you may instead use the Color QuickDraw procedure `RGBForeColor`, which is described in the chapter "Color QuickDraw."

BackColor

To change a basic graphics port's background color, use the `BackColor` procedure.

```
PROCEDURE BackColor (color: LongInt);
```

`color` One of eight color values. You can use the constants described for the `ForeColor` procedure in the previous section.

DESCRIPTION

The `BackColor` procedure sets the background color for the current graphics port to the color that you specify in the `color` parameter. When you draw with the `patCopy` and `srcCopy` transfer modes, for example, white pixels are drawn in the color you specify with `BackColor`.

SPECIAL CONSIDERATIONS

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

SEE ALSO

All nonwhite colors appear as black on black-and-white screens. Before you use `BackColor`, you can use the `DeviceLoop` procedure, which is described in the chapter "Graphics Devices," to determine the color characteristics of the current screen.

In System 7, you may instead use the Color QuickDraw procedure `RGBBackColor`, which is described in the chapter "Color QuickDraw."

ColorBit

Use the `ColorBit` procedure to set the foreground color for all printing in the current graphics port.

```
PROCEDURE ColorBit (whichBit: Integer);
```

`whichBit` An integer specifying the plane to draw into.

DESCRIPTION

The `ColorBit` procedure is called by printing software for a color printer (or other color-imaging software) to set the `GrafPort` record's `colorBit` field to the value in the `whichBit` parameter. This value tells QuickDraw which plane of the color picture to draw into. QuickDraw draws into the plane corresponding to the bit number specified by the `whichBit` parameter. Since QuickDraw can support output devices that have up to 32 bits of color information per pixel, the possible range of values for `whichBit` is 0 through 31. The initial value of the `colorBit` field is 0.

Determining Whether QuickDraw Has Finished Drawing

Your application can use the `QDDone` function to determine whether drawing is completed in a given graphics port. You can also use it to determine whether drawing has finished in all open graphics ports.

QDDone

Although you will probably never need to determine whether QuickDraw has completed drawing, you can do so by using the `QDDone` function.

```
FUNCTION QDDone (port: GrafPtr): Boolean;
```

`port` The `GrafPort` record for a graphics port in which your application has begun drawing; if you pass `NIL`, `QDDone` tests all open graphics ports.

DESCRIPTION

The `QDDone` function returns `TRUE` if all drawing operations have finished in the graphics port specified in the `port` parameter, `FALSE` if any remain to be executed. If you pass `NIL` in the `port` parameter, then `QDDone` returns `TRUE` only if drawing operations have completed in all ports.

The `QDDone` function may be useful if a graphics accelerator is present and operating asynchronously. You could use it to ensure that all drawing is done before issuing new drawing commands, and to avoid the possibility that the new drawing operations might be overlaid by previously issued but unexecuted operations.

SPECIAL CONSIDERATIONS

The `QDDone` function has little or no usefulness.

If a graphics port draws a clock or some other continuously operating drawing process, `QDDone` may never return `TRUE`.

To determine whether all drawing in a color graphics port has completed, you must coerce its `CGrafPort` record to a `GrafPort` record, which you pass in the `port` parameter.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `QDDone` function are

Trap macro	Selector
`_QDExtensions`	$00040013

Getting Pattern Resources

As described in "Bit Patterns" beginning on page 3-5, QuickDraw predefines five patterns for your use in the global variables `white`, `black`, `gray`, `ltGray`, and `dkGray`. However, you can create and store your own patterns in a resource file. To retrieve the patterns stored in a pattern (`'PAT '`) resource, you can use the `GetPattern` function. To retrieve the patterns stored in a pattern list (`'PAT#'`) resource, you can use the `GetIndPattern` procedure.

GetPattern

To get a pattern (`'PAT '`) resource stored in a resource file, you can use the `GetPattern` function.

```
FUNCTION GetPattern (patID: Integer): PatHandle;
```

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

DESCRIPTION

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

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

If a pattern resource with the ID that you request does not exist, the `GetPattern` function returns `NIL`.

The data structure of type `PatHandle` is defined as follows:

```
TYPE  PatPrt      = ^Pattern;
      PatHandle   = ^PatPtr;
      Pattern = PACKED ARRAY[0..7] OF 0..255;
```

When you are finished using the pattern, dispose of its handle with the Memory Manager function `DisposeHandle`.

SPECIAL CONSIDERATIONS

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

SEE ALSO

The pattern resource is described on page 3-140; the `Pattern` record is described on page 3-40. See the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information about resources, the Resource Manager, and the `GetResource` function. See *Inside Macintosh: Memory* for information about the `DisposeHandle` procedure.

GetIndPattern

To get a pattern stored in a pattern list (`'PAT#'`) resource, you can use the `GetIndPattern` procedure.

```
PROCEDURE GetIndPattern (VAR thePattern: Pattern;
                         patListID: Integer; index: Integer);
```

`thePattern` A `Pattern` record.

`patListID` The resource ID for a resource of type `'PAT#'`.

`index` The index number for the desired pattern within the pattern list (`'PAT#'`) resource.

DESCRIPTION

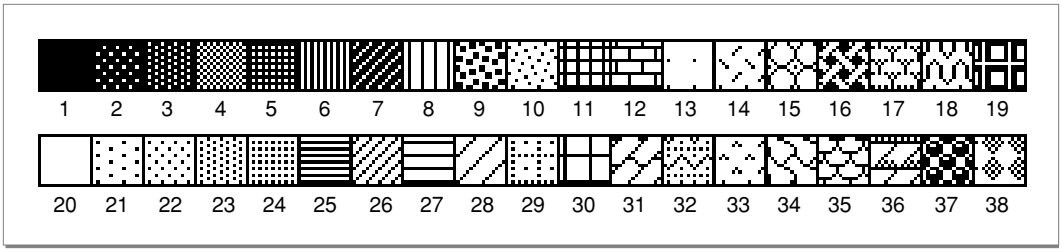
In the parameter `thePattern`, the `GetIndPattern` procedure returns a `Pattern` record for a pattern stored in a pattern list (`'PAT#'`) resource. Specify the resource ID for a pattern list (`'PAT#'`) resource in the `patListID` parameter. In the `index` parameter, specify the index number to a particular pattern stored in that resource. The index number can range from 1 to the number of patterns in the pattern list resource. The `GetIndPattern` procedure calls the following Resource Manager function with these parameters:

```
GetResource('PAT ', patListID);
```

There is a pattern list resource in the System file that contains the standard Macintosh patterns used by MacPaint. Figure 3-28 shows these standard patterns. The resource ID, and the constant you can use to represent it, are

```
CONST sysPatListID = 0;
```

Figure 3-28 Standard patterns



SPECIAL CONSIDERATIONS

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

SEE ALSO

The pattern list resource is described on page 3-141; the `Pattern` record is described on page 3-40. See the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information about resources, the Resource Manager, and the `GetResource` function.

Customizing QuickDraw Operations

For each shape that QuickDraw can draw, there are procedures that perform these basic graphics operations on the shape: frame, paint, erase, invert, and fill. Those procedures in turn call a low-level drawing routine for the shape. For example, the `FrameOval`, `PaintOval`, `EraseOval`, `InvertOval`, and `FillOval` procedures all call the low-level procedure `StdOval`, which draws the oval.

Other low-level routines defined by QuickDraw are:

- The procedure called by `CopyBits` that performs bit and pixel transfer.
- The function that measures the width of text and is called by the QuickDraw text routines `CharWidth`, `StringWidth`, and `TextWidth`. (These QuickDraw text routines are described in the chapter "QuickDraw Text" in *Inside Macintosh: Text*.)
- The procedure that processes picture comments. The standard procedure ignores picture comments. (Picture comments are described in Appendix B.)
- The procedure that saves drawing commands as the definition of a picture, and the procedure that retrieves them. These two enable your application to draw on remote devices, print to the disk, get picture input from the disk, and support large pictures.

For each type of object QuickDraw can draw, including text and lines, there's a pointer to one of these low-level routines.

The `grafProcs` field of a `GrafPort` or `CGrafPort` record determines which low-level routines are called. If that field contains the value of `NIL`, the standard routines are called. You can set the `grafProcs` field to point to a record of pointers to your own routines. For a basic graphics port, this record of pointers is defined by a `QDProcs` record. As described in the chapter "Color QuickDraw," these pointers are contained in a `CQDProcs` record for a color graphics port. By changing these pointers, you can install your own routines, and either completely override the standard ones or call them after your routines have modified their parameters as necessary.

To assist you in setting up a record, basic QuickDraw provides the `SetStdProcs` procedure, which is described in the next section. You can use the `SetStdProcs` procedure to get a `QDProcs` record with fields that point to basic QuickDraw's standard low-level routines. You can then reset the routines with which you are concerned. By pointing to your modified `QDProcs` record in the `grafProcs` field of a `GrafPort` record, you can replace some or all of basic QuickDraw's standard low-level routines.

The standard QuickDraw low-level routines are described in the rest of this section. These low-level routines should be called only from your customized routines.

SetStdProcs

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

```
PROCEDURE SetStdProcs (VAR procs: QDProcs);
```

`procs` Upon completion, a `QDProcs` record with fields that point to basic QuickDraw's standard low-level routines.

DESCRIPTION

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

The routines you install in this `QDProcs` record must have the same calling sequences as the standard routines, which are described in the rest of this section.

SPECIAL CONSIDERATIONS

The Color QuickDraw procedure `SetStdCProcs` is analogous to the `SetStdProcs` procedure, which you should use with computers that support only basic QuickDraw. When drawing in a color graphics port, your application must always use `SetStdCProcs` instead of `SetStdProcs`.

SEE ALSO

The data structure of type `QDProcs` is described on page 3-39. The `SetStdCProcs` procedure is described in the chapter "Color QuickDraw."

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

StdText

The `StdText` procedure is QuickDraw's standard low-level routine for drawing text.

```
PROCEDURE StdText (byteCount: Integer; textBuf: Ptr;
                   numer,denom: Point);
```

`byteCount`	The number of bytes of text to draw.
`textBuf`	A memory structure containing the text to draw.
`numer`	Scaling numerator.
`denom`	Scaling denominator.

DESCRIPTION

The `StdText` procedure draws text from the arbitrary structure in memory specified by the `textBuf` parameter, starting from the first byte and continuing for the number of bytes specified in the `byteCount` parameter. The `numer` and `denom` parameters specify the scaling factor: `numer.v` over `denom.v` gives the vertical scaling, and `numer.h` over `denom.h` gives the horizontal scaling factor.

SPECIAL CONSIDERATIONS

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

SEE ALSO

QuickDraw's text-drawing capabilities are described in the chapter "QuickDraw Text" in *Inside Macintosh: Text*.

StdLine

The `StdLine` procedure is QuickDraw's standard low-level routine for drawing a line.

```
PROCEDURE StdLine (newPt: Point);
```

`newPt` The point to which to draw the line.

DESCRIPTION

The `StdLine` procedure draws a line from the current pen location to the location (in local coordinates) specified in the `newPt` parameter.

SPECIAL CONSIDERATIONS

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

StdRect

The `StdRect` procedure is QuickDraw's standard low-level routine for drawing a rectangle.

```
PROCEDURE StdRect (verb: GrafVerb; r: Rect);
```

`verb` One of the following actions to perform, as defined for the `GrafVerb` data type:

```
GrafVerb = (frame, paint, erase, invert, fill);
```

`r` The rectangle to draw.

DESCRIPTION

The `StdRect` procedure draws the rectangle specified in the `r` parameter according to the action specified in the `verb` parameter.

SPECIAL CONSIDERATIONS

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

StdRRect

The `StdRRect` procedure is QuickDraw's standard low-level routine for drawing a rounded rectangle.

```
PROCEDURE StdRRect (verb: GrafVerb; r: Rect;
                    ovalwidth,ovalHeight: Integer)
```

`verb` One of the following actions to perform, as defined for the `GrafVerb` data type:

```
GrafVerb = (frame, paint, erase, invert, fill);
```

`r` The rectangle to draw.

`ovalwidth` The width diameter for the corner oval.

`ovalHeight` The height diameter for the corner oval.

DESCRIPTION

The `StdRRect` procedure draws the rounded rectangle specified in the `r` parameter according to the action specified in the `verb` parameter. The `ovalWidth` and `ovalHeight` parameters specify the diameters of curvature for the corners.

SPECIAL CONSIDERATIONS

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

StdOval

The `StdOval` procedure is QuickDraw's standard low-level routine for drawing an oval.

```
PROCEDURE StdOval (verb: GrafVerb; r: Rect);
```

`verb` One of the following actions to perform, as defined for the `GrafVerb` data type:

```
GrafVerb = (frame, paint, erase, invert, fill);
```

`r` The rectangle to contain the oval.

DESCRIPTION

The `StdOval` procedure draws an oval inside the given rectangle specified in the `r` parameter according to the action specified in the `verb` parameter.

SPECIAL CONSIDERATIONS

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

StdArc

The `StdArc` procedure is QuickDraw's standard low-level routine for drawing an arc or a wedge.

```
PROCEDURE StdArc (verb: GrafVerb; r: Rect;
                  startAngle,arcAngle: Integer);
```

`verb` One of the following actions to perform, as defined for the `GrafVerb` data type:

```
GrafVerb = (frame, paint, erase, invert, fill);
```

`r` The rectangle to contain the arc.

`startAngle` The beginning angle.

`arcAngle` The ending angle.

DESCRIPTION

Using the action specified in the `verb` parameter, the `StdArc` procedure draws an arc or wedge of the oval that fits inside the rectangle specified in the `r` parameter. The arc or wedge is bounded by the radii specified in the `startAngle` and `arcAngle` parameters. (The `startAngle` and `arcAngle` parameters are illustrated in Figure 3-20 on page 3-72.)

SPECIAL CONSIDERATIONS

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

StdPoly

The `StdPoly` procedure is QuickDraw's standard low-level routine for drawing a polygon.

```
PROCEDURE StdPoly (verb: GrafVerb; poly: PolyHandle);
```

`verb` One of the following actions to perform, as defined for the `GrafVerb` data type:

```
GrafVerb = (frame, paint, erase, invert, fill);
```

`poly` A handle to the polygon data.

DESCRIPTION

The `StdPoly` procedure draws the polygon specified in the `poly` parameter according to the action specified in the `verb` parameter.

SPECIAL CONSIDERATIONS

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

StdRgn

The `StdRgn` procedure is QuickDraw's standard low-level routine for drawing a region.

```
PROCEDURE StdRgn (verb: GrafVerb; rgn: RgnHandle);
```

`verb` One of the following actions to perform, as defined for the `GrafVerb` data type:

```
GrafVerb = (frame, paint, erase, invert, fill);
```

`rgn` A handle to the region data.

DESCRIPTION

The `StdRgn` procedure draws the region specified in the `rgn` parameter according to the action specified in the `verb` parameter.

SPECIAL CONSIDERATIONS

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

StdBits

The `StdBits` procedure is QuickDraw's standard low-level routine for doing bit and pixel transfer.

```
PROCEDURE StdBits (VAR srcBits: BitMap; VAR srcRect,dstRect: Rect;
                   mode: Integer; maskRgn: RgnHandle);
```

`srcBits`	A bitmap or pixel map containing the image to copy.
`srcRect`	The source rectangle.
`dstRect`	The destination rectangle.
`mode`	The source mode for the copy.
`maskRgn`	A handle to a region acting as a mask for the transfer.

DESCRIPTION

The `StdBits` procedure transfers a bit or pixel image between the bitmap or pixel map specified in the `srcBits` parameter and bitmap of the current graphics port, just as if the `CopyBits` procedure were called with the same parameters and with a destination bitmap equal to `thePort^.portBits`.

SPECIAL CONSIDERATIONS

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

SEE ALSO

See the description of the `CopyBits` procedure beginning on page 3-112 for a discussion of the destination bitmap and of the `srcBits`, `srcRect`, `dstRect`, `mode`, and `maskRgn` parameters.

StdComment

The `StdComment` procedure is QuickDraw's standard low-level routine for processing a picture comment.

```
PROCEDURE StdComment (kind,dataSize: Integer;
                      dataHandle: Handle);
```

`kind` The type of comment. See Appendix A in this book for a list of the standard constants (and the values they represent) used to specify common picture comment types.

`dataSize` The size of additional data.

`dataHandle` A handle to additional data.

DESCRIPTION

The `kind` parameter identifies the type of comment. The `dataHandle` parameter takes a handle to additional data, and the `dataSize` parameter specifies the size of that data in bytes. If there's no additional data for the comment, the value of the `dataHandle` parameter is `NIL` and the value of the `dataSize` parameter is 0. The `StdComment` procedure simply ignores the comment.

SPECIAL CONSIDERATIONS

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

SEE ALSO

Picture comments are described in detail in Appendix B, "Using Picture Comments for Printing."

StdTxtMeas

The `StdTxtMeas` function is QuickDraw's standard low-level routine for measuring text width.

```
FUNCTION StdTxtMeas (byteCount: Integer; textAddr: Ptr;
                     VAR numer,denom: Point;
                     VAR info: FontInfo): Integer;
```

`byteCount`	The number of text bytes to measure.
`textAddr`	A pointer to the memory structure containing the text.
`numer`	Scaling numerator.
`denom`	Scaling denominator.
`info`	A `FontInfo` record.

DESCRIPTION

The `StdTxtMeas` function returns the width of the text stored in the arbitrary structure in memory specified by `textAddr`, starting with the first byte and continuing for `byteCount` bytes. The `numer` and `denom` parameters specify the scaling as in the `StdText` procedure; note that `StdTxtMeas` may change them.

SPECIAL CONSIDERATIONS

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

SEE ALSO

QuickDraw's text-drawing capabilities are described in the chapter "QuickDraw Text" in *Inside Macintosh: Text*.

StdGetPic

The `StdGetPic` procedure is QuickDraw's standard low-level routine for retrieving information from the definition of a picture.

```
PROCEDURE StdGetPic (dataPtr: Ptr; byteCount: Integer);
```

`dataPtr`	A pointer to the collected picture data.
`byteCount`	The size of the picture data.

DESCRIPTION

The `StdGetPic` procedure retrieves from the definition of the currently open picture the next number of bytes as specified in the `byteCount` parameter. The `StdGetPic` procedure stores them in the data structure pointed to by the `dataPtr` parameter.

SEE ALSO

Pictures are described in the chapter "Pictures," which also provides a code sample illustrating how you can supply your application with its own low-level procedure for retrieving pictures.

StdPutPic

The `StdPutPic` procedure is QuickDraw's standard low-level routine for saving information as the definition of a picture.

```
PROCEDURE StdPutPic (dataPtr: Ptr; byteCount: Integer);
```

`dataPtr` A pointer to the collected picture data.

`byteCount` The size of the picture data.

DESCRIPTION

The `StdPutPic` procedure saves as the definition of the currently open picture the drawing commands stored in the data structure pointed to by the `dataPtr` parameter, starting with the first byte and continuing for the next number of bytes as specified in the `byteCount` parameter.

SPECIAL CONSIDERATIONS

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

SEE ALSO

Pictures are described in the chapter "Pictures," which also provides a code sample illustrating how you can supply your application with its own low-level procedure for saving pictures.

Resources

This section describes the resources you can create to define bit patterns for use when drawing, painting, or filling. The pattern (`'PAT '`) resource defines a single bit pattern. The pattern list (`'PAT#'`) resource defines an array of bit patterns.

A bit pattern is a 64-bit image, organized as an 8-by-8 pixel square, that defines a repeating design or tone. (Resources for color pixel patterns are described in the chapter "Color QuickDraw.")

This section describes the structures of these resources after they are compiled by the Rez resource compiler, available from APDA. To create pattern and pattern list resources, you typically use a high-level tool such as the ResEdit application. You can then use the DeRez decompiler to convert your resources into Rez input when necessary.

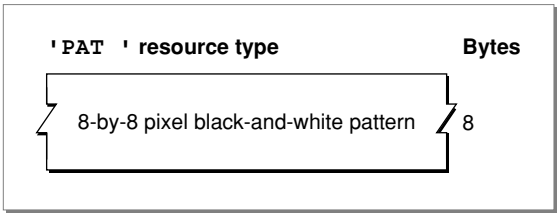
The Pattern Resource

You can use a pattern resource to define a single bit pattern. A pattern resource is a resource of type `'PAT '`. All pattern resources that you create must have resource ID numbers greater than 128.

To retrieve the bit pattern stored in a pattern resource, you can use the `GetPattern` function, which is described on page 3-126. You can then specify that bit pattern for a fill pattern, background pattern, or pen pattern.

A pattern resource is defined to be of type `hex String[8];` every bit represents a pixel in the 8-by-8 pixel pattern. If you examine the compiled version of a pattern resource, as represented in Figure 3-29, you find that it contains 8 bytes of information that define the 8-by-8 pixel square of the pattern.

Figure 3-29 Format of a compiled pattern (`'PAT '`) resource



The Pattern List Resource

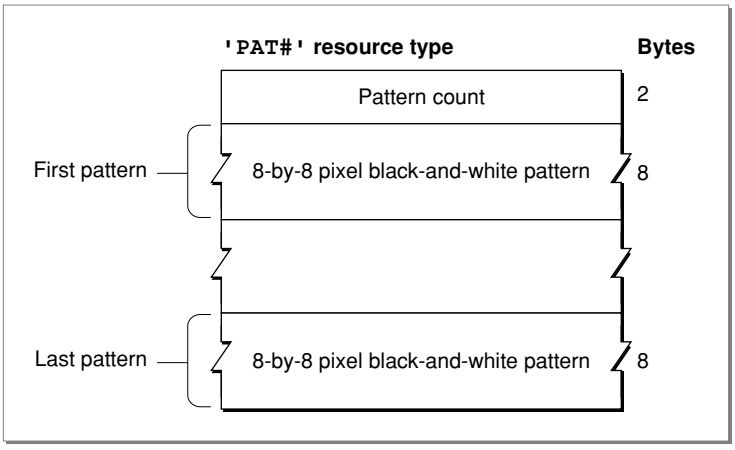
You can use a pattern list resource to define an array of bit patterns. A pattern list resource is a resource of type `'PAT#'`. All pattern list resources that you create must have resource ID numbers greater than 128.

To retrieve one of the bit patterns stored in a pattern list resource, you can use the `GetIndPattern` procedure, which is described on page 3-127. You can then specify that bit pattern for a fill pattern, background pattern, or pen pattern.

If you examine the compiled version of a pattern list resource, as represented in Figure 3-30, you find that it contains the following information:

- Pattern count. This is the number of bit patterns defined in this resource.
- An array of bit patterns, each of which contains 8 bytes of information that define the 8-by-8 pixel square of the pattern.

Figure 3-30 Format of a compiled pattern list (`'PAT#'`) resource



Summary of QuickDraw Drawing

Pascal Summary

Constants

```
CONST
   {basic QuickDraw colors}
   whiteColor     = 30;
   blackColor     = 33;
   yellowColor    = 69;
   magentaColor   = 137;
   redColor       = 205;
   cyanColor      = 273;
   greenColor     = 341;
   blueColor      = 409;

   {source modes for basic graphics ports}
   srcCopy        = 0;  {where source pixel is black, force destination }
                        { pixel black; where source pixel is white, force }
                        { destination pixel white}
   srcOr          = 1;  {where source pixel is black, force destination }
                        { pixel black; where source pixel is white, leave }
                        { destination pixel unaltered}
   srcXor         = 2;  {where source pixel is black, invert destination }
                        { pixel; where source pixel is white, leave }
                        { destination pixel unaltered}
   srcBic         = 3;  {where source pixel is black, force destination }
                        { pixel white; where source pixel is white, leave }
                        { destination pixel unaltered}
   notSrcCopy     = 4;  {where source pixel is black, force destination }
                        { pixel white; where source pixel is white, force }
                        { destination pixel black}
   notSrcOr       = 5;  {where source pixel is black, leave destination }
                        { pixel unaltered; where source pixel is white, }
                        { force destination pixel black}
```

```
notSrcXor       = 6;  {where source pixel is black, leave destination }
                      { pixel unaltered; where source pixel is white, }
                      { invert destination pixel}
notSrcBic       = 7;  {where source pixel is black, leave destination }
                      { pixel unaltered; where source pixel is white, }
                      { force destination pixel white}

{pattern modes}
patCopy         = 8;  {where pattern pixel is black, apply foreground }
                      { color to destination pixel; where pattern pixel }
                      { is white, apply background color to destination }
                      { pixel}
patOr           = 9;  {where pattern pixel is black, invert destination }
                      { pixel; where pattern pixel is white, leave }
                      { destination pixel unaltered}
patXor          = 10; {where pattern pixel is black, invert destination }
                      { pixel; where pattern pixel is white, leave }
                      { destination pixel unaltered}
patBic          = 11; {where pattern pixel is black, apply background }
                      { color to destination pixel; where pattern pixel }
                      { is white, leave destination pixel unaltered}
notPatCopy      = 12; {where pattern pixel is black, apply background }
                      { color to destination pixel; where pattern pixel }
                      { is white, apply foreground color to destination }
                      { pixel}
notPatOr        = 13; {where pattern pixel is black, leave destination }
                      { pixel unaltered; where pattern pixel is white, }
                      { apply foreground color to destination pixel}
notPatXor       = 14; {where pattern pixel is black, leave destination }
                      { pixel unaltered; where pattern pixel is white, }
                      { invert destination pixel}
notPatBic       = 15; {where pattern pixel is black, leave destination }
                      { pixel unaltered; where pattern pixel is white, }
                      { apply background color to destination pixel}

ditherCopy      = 64; {add to source mode for dithering}

{pattern list resource ID for patterns in the System file}
sysPatListID = 0;
```

Data Types

```
TYPE PolyPtr   = ^Polygon;
PolyHandle     = ^PolyPtr;
Polygon        =
RECORD
   polySize:   Integer;                {size in bytes}
   polyBBox:   Rect;                   {bounding rectangle}
   polyPoints: ARRAY[0..0] OF Point;   {vertices for polygon}
END;

PenState =
RECORD
   pnLoc:   Point;      {pen location}
   pnSize:  Point;      {pen size}
   pnMode:  Integer;    {pen's pattern mode}
   pnPat:   Pattern;    {pen pattern}
END;

QDProcsPtr     = ^QDProcs;
QDProcs        =
RECORD
   textProc:      Ptr;  {text drawing}
   lineProc:      Ptr;  {line drawing}
   rectProc:      Ptr;  {rectangle drawing}
   rRectProc:     Ptr;  {roundRect drawing}
   ovalProc:      Ptr;  {oval drawing}
   arcProc:       Ptr;  {arc/wedge drawing}
   rgnProc:       Ptr;  {region drawing}
   bitsProc:      Ptr;  {bit transfer}
   commentProc:   Ptr;  {picture comment processing}
   txMeasProc:    Ptr;  {text width measurement}
   getPicProc:    Ptr;  {picture retrieval}
   putPicProc:    Ptr;  {picture saving}
END;

GrafVerb = (frame,paint,erase,invert,fill);

PatPtr            = ^Pattern;
PatHandle         = ^PatPtr;
Pattern           = PACKED ARRAY[0..7] OF 0..255;
```

Routines

Managing the Graphics Pen

```
PROCEDURE HidePen;
PROCEDURE ShowPen;
PROCEDURE GetPen            (VAR pt: Point);
PROCEDURE GetPenState       (VAR pnState: PenState);
PROCEDURE SetPenState       (pnState: PenState);
PROCEDURE PenSize           (width,height: Integer);
PROCEDURE PenMode           (mode: Integer);
PROCEDURE PenPat            (pat: Pattern);
PROCEDURE PenNormal;
```

Changing the Background Bit Pattern

```
PROCEDURE BackPat           (pat: Pattern);
```

Drawing Lines

```
PROCEDURE MoveTo            (h,v: Integer);
PROCEDURE Move              (dh,dv: Integer);
PROCEDURE LineTo            (h,v: Integer);
PROCEDURE Line              (dh,dv: Integer);
```

Creating and Managing Rectangles

```
PROCEDURE SetRect           (VAR r: Rect; left,top,right,bottom: Integer);
PROCEDURE OffsetRect        (VAR r: Rect; dh,dv: Integer);
PROCEDURE InsetRect         (VAR r: Rect; dh,dv: Integer);
FUNCTION SectRect           (src1,src2: Rect; VAR dstRect: Rect): Boolean;
PROCEDURE UnionRect         (src1,src2: Rect; VAR dstRect: Rect);
FUNCTION PtInRect           (pt: Point; r: Rect): Boolean;
PROCEDURE Pt2Rect           (pt1,pt2: Point; VAR dstRect: Rect);
PROCEDURE PtToAngle         (r: Rect; pt: Point; VAR angle: Integer);
FUNCTION EqualRect          (rect1,rect2: Rect): Boolean;
FUNCTION EmptyRect          (r:  Rect): Boolean;
```

Drawing Rectangles

```
PROCEDURE FrameRect          (r: Rect);
PROCEDURE PaintRect          (r: Rect);
PROCEDURE FillRect           (r: Rect; pat: Pattern);
PROCEDURE EraseRect          (r: Rect);
PROCEDURE InvertRect         (r: Rect);
```

Drawing Rounded Rectangles

```
PROCEDURE FrameRoundRect     (r: Rect; ovalWidth,ovalHeight: Integer);
PROCEDURE PaintRoundRect     (r: Rect; ovalWidth,ovalHeight: Integer);
PROCEDURE FillRoundRect      (r: Rect; ovalWidth,ovalHeight: Integer;
                              pat: Pattern);
PROCEDURE EraseRoundRect     (r: Rect; ovalWidth,ovalHeight: Integer);
PROCEDURE InvertRoundRect    (r: Rect; ovalWidth,ovalHeight: Integer);
```

Drawing Ovals

```
PROCEDURE FrameOval          (r: Rect);
PROCEDURE PaintOval          (r: Rect);
PROCEDURE FillOval           (r: Rect; pat: Pattern);
PROCEDURE EraseOval          (r: Rect);
PROCEDURE InvertOval         (r: Rect);
```

Drawing Arcs and Wedges

```
PROCEDURE FrameArc           (r: Rect; startAngle,arcAngle: Integer);
PROCEDURE PaintArc           (r: Rect; startAngle,arcAngle: Integer);
PROCEDURE FillArc            (r: Rect; startAngle,arcAngle: Integer;
                              pat: Pattern);
PROCEDURE EraseArc           (r: Rect; startAngle,arcAngle: Integer);
PROCEDURE InvertArc          (r: Rect; startAngle,arcAngle: Integer);
```

Creating and Managing Polygons

```
FUNCTION OpenPoly            : PolyHandle;
PROCEDURE ClosePoly;
PROCEDURE OffsetPoly         (poly: PolyHandle; dh,dv: Integer);
PROCEDURE KillPoly           (poly: PolyHandle);
```

Drawing Polygons

```
PROCEDURE FramePoly         (poly: PolyHandle);
PROCEDURE PaintPoly         (poly: PolyHandle);
PROCEDURE FillPoly          (poly: PolyHandle; pat: Pattern);
PROCEDURE ErasePoly         (poly: PolyHandle);
PROCEDURE InvertPoly        (poly: PolyHandle);
```

Creating and Managing Regions

```
FUNCTION NewRgn             : RgnHandle;
PROCEDURE OpenRgn;
PROCEDURE CloseRgn          (dstRgn: rgnHandle);
PROCEDURE DisposeRgn        (rgn: RgnHandle);
PROCEDURE CopyRgn           (srcRgn,dstRgn: RgnHandle);
PROCEDURE SetEmptyRgn       (rgn: RgnHandle);
PROCEDURE SetRectRgn        (rgn: RgnHandle;
                             left,top,right,bottom: Integer);
PROCEDURE RectRgn           (rgn: RgnHandle; r: Rect);
PROCEDURE OffsetRgn         (rgn: RgnHandle; dh,dv: Integer);
PROCEDURE InsetRgn          (rgn: RgnHandle; dh,dv: Integer);
PROCEDURE SectRgn           (srcRgnA,srcRgnB,dstRgn: RgnHandle);
PROCEDURE UnionRgn          (srcRgnA,srcRgnB,dstRgn: RgnHandle);
PROCEDURE DiffRgn           (srcRgnA,srcRgnB,dstRgn: RgnHandle);
PROCEDURE XorRgn            (srcRgnA,srcRgnB,dstRgn: RgnHandle);
FUNCTION PtInRgn            (pt: Point; rgn: RgnHandle): Boolean;
FUNCTION RectInRgn          (r: Rect; rgn: RgnHandle): Boolean;
FUNCTION EqualRgn           (rgnA,rgnB: RgnHandle): Boolean;
FUNCTION EmptyRgn           (rgn: RgnHandle): Boolean;
```

Drawing Regions

```
PROCEDURE FrameRgn          (rgn: RgnHandle);
PROCEDURE PaintRgn          (rgn: RgnHandle);
PROCEDURE FillRgn           (rgn: RgnHandle; pat: Pattern);
PROCEDURE EraseRgn          (rgn: RgnHandle);
PROCEDURE InvertRgn         (rgn: RgnHandle);
```

Scaling and Mapping Points, Rectangles, Polygons, and Regions

```
PROCEDURE ScalePt           (VAR pt: Point; srcRect,dstRect: Rect);
PROCEDURE MapPt             (VAR pt: Point; srcRect,dstRect: Rect);
PROCEDURE MapRect           (VAR r: Rect; srcRect,dstRect: Rect);
PROCEDURE MapRgn            (rgn: RgnHandle; srcRect,dstRect: Rect);
PROCEDURE MapPoly           (poly: PolyHandle; srcRect,dstRect: Rect);
```

Calculating Black-and-White Fills

```
PROCEDURE SeedFill          (srcPtr,dstPtr: Ptr;
                             srcRow,dstRow,height,words,
                             seedH,seedV: Integer);
PROCEDURE CalcMask          (srcPtr,dstPtr: Ptr;
                             srcRow,dstRow,height,words: Integer);
```

Copying Images

```
PROCEDURE CopyBits          (srcBits,dstBits: BitMap; srcRect,dstRect: Rect;
                             mode: Integer; maskRgn: RgnHandle);
PROCEDURE CopyMask          (srcBits,maskBits,dstBits: BitMap;
                             srcRect,maskRect,dstRect: Rect);
PROCEDURE CopyDeepMask      (srcBits: BitMap; maskBits: BitMap;
                             dstBits: BitMap; srcRect: Rect;
                             maskRect: Rect; dstRect: Rect;
                             mode: Integer; maskRgn: RgnHandle);
```

Drawing With the Eight-Color System

```
PROCEDURE ForeColor         (color: LongInt);
PROCEDURE BackColor         (color: LongInt);
PROCEDURE ColorBit          (whichBit: Integer);
```

Determining Whether QuickDraw Has Finished Drawing

```
FUNCTION QDDone             (port: GrafPtr): Boolean;
```

Getting Pattern Resources

```
FUNCTION GetPattern         (patID: Integer): PatHandle;
PROCEDURE GetIndPattern     (VAR thePattern: Pattern; patListID: Integer;
                             index: Integer);
```

Customizing QuickDraw Operations

```
PROCEDURE SetStdProcs       (VAR procs: QDProcs);
PROCEDURE StdText           (byteCount: Integer; textBuf: Ptr;
                             numer,denom: Point);
PROCEDURE StdLine           (newPt: Point);
PROCEDURE StdRect           (verb: GrafVerb; r: Rect);
PROCEDURE StdRRect          (verb: GrafVerb; r: Rect;
                             ovalwidth,ovalHeight: Integer);
PROCEDURE StdOval           (verb: GrafVerb; r: Rect);
PROCEDURE StdArc            (verb: GrafVerb; r: Rect;
                             startAngle,arcAngle: Integer);
PROCEDURE StdPoly           (verb: GrafVerb; poly: PolyHandle);
PROCEDURE StdRgn            (verb: GrafVerb; rgn: RgnHandle);
PROCEDURE StdBits           (VAR srcBits: BitMap;
                             VAR srcRect,dstRect: Rect; mode: Integer;
                             maskRgn: RgnHandle);
PROCEDURE StdComment        (kind,dataSize: Integer; dataHandle: Handle);
FUNCTION StdTxtMeas         (byteCount: Integer; textAddr: Ptr;
                             VAR numer, denom: Point;
                             VAR info: FontInfo): Integer;
PROCEDURE StdGetPic         (dataPtr: Ptr; byteCount: Integer);
PROCEDURE StdPutPic         (dataPtr: Ptr; byteCount: Integer);
```

C Summary

Constants

```
enum {
   /* basic QuickDraw colors */
   whiteColor     = 30;
   blackColor     = 33;
   yellowColor    = 69;
   magentaColor   = 137;
   redColor       = 205;
   cyanColor      = 273;
   greenColor     = 341;
   blueColor      = 409;
```

```
/* source modes */
srcCopy         = 0,  /* where source pixel is black, force destination
                         pixel black; where source pixel is white, force
                         destination pixel white */
srcOr           = 1,  /* where source pixel is black, force destination
                         pixel black; where source pixel is white, leave
                         destination pixel unaltered */
srcXor          = 2,  /* where source pixel is black, invert destination
                         pixel; where source pixel is white, leave
                         destination pixel unaltered */
srcBic          = 3,  /* where source pixel is black, force destination
                         pixel white; where source pixel is white, leave
                         destination pixel unaltered */
notSrcCopy      = 4,  /* where source pixel is black, force destination
                         pixel white; where source pixel is white, force
                         destination pixel black */
notSrcOr        = 5,  /* where source pixel is black, leave destination
                         pixel unaltered; where source pixel is white,
                         force destination pixel black */
notSrcXor       = 6,  /* where source pixel is black, leave destination
                         pixel unaltered; where source pixel is white,
                         invert destination pixel*/
notSrcBic       = 7,  /* where source pixel is black, leave destination
                         pixel unaltered; where source pixel is white,
                         force destination pixel white */

/* pattern modes */
patCopy         = 8,  /* where pattern pixel is black, apply foreground
                         color to destination pixel; where pattern pixel
                         is white, apply background color to destination
                         pixel */
patOr           = 9,  /* where pattern pixel is black, invert destination
                         pixel; where pattern pixel is white, leave
                         destination pixel unaltered */
patXor          = 10; /* where pattern pixel is black, invert destination
                         pixel; where pattern pixel is white, leave
                         destination pixel unaltered */
patBic          = 11; /* where pattern pixel is black, apply background
                         color to destination pixel; where pattern pixel
                         is white, leave destination pixel unaltered */
```

```
   notPatCopy     = 12; /* where pattern pixel is black, apply background
                           color to destination pixel; where pattern pixel
                           is white, apply foreground color to destination
                           pixel */
   notPatOr       = 13; /* where pattern pixel is black, leave destination
                           pixel unaltered; where pattern pixel is white,
                           apply foreground color to destination pixel */
   notPatXor      = 14; /* where pattern pixel is black, leave destination
                           pixel unaltered; where pattern pixel is white,
                           invert destination pixel */
   notPatBic      = 15; /* where pattern pixel is black, leave destination
                           pixel unaltered; where pattern pixel is white,
                           apply background color to destination pixel */

   ditherCopy     = 64, /* add to source mode for dithering */

   /* pattern list resource ID for patterns in the System file */
   sysPatListID = 0
};
```

Data Types

```
struct Polygon {
   short polySize;      /* size in bytes */
   Rect  polyBBox;      /* bounding rectangle */
   Point polyPoints[1]; /* vertices for polygon */
};
typedef struct Polygon Polygon;
typedef Polygon *PolyPtr, **PolyHandle;

struct PenState {
   Point    pnLoc;      /* pen location */
   Point    pnSize;     /* pen size */
   short    pnMode;     /* pen's pattern mode */
   Pattern  pnPat;      /* pen pattern */
};
typedef struct PenState PenState;

struct QDProcs {
   Ptr textProc;        /* text drawing */
   Ptr lineProc;        /* line drawing */
   Ptr rectProc;        /* rectangle drawing */
   Ptr rRectProc;       /* roundRect drawing */
   Ptr ovalProc;        /* oval drawing */
```

```
   Ptr arcProc;         /* arc and wedge drawing */
   Ptr polyProc;        /* region drawing */
   Ptr rgnProc;         /* region drawing */
   Ptr bitsProc;        /* bit transfer */
   Ptr commentProc;     /* picture comment processing */
   Ptr txMeasProc;      /* text width measurement */
   Ptr getPicProc;      /* picture retrieval */
   Ptr putPicProc;      /* picture saving */
};
typedef struct QDProcs QDProcs;
typedef QDProcs *QDProcsPtr;

enum {frame,paint,erase,invert,fill};
typedef unsigned char GrafVerb;

struct Pattern{
   unsigned char pat[8];
};
typedef struct Pattern Pattern;
typedef Pattern *PatPtr;
typedef const unsigned char *ConstPatternParam;
typedef PatPtr *PatHandle;
```

Functions

Managing the Graphics Pen

```
pascal void HidePen         (void);
pascal void ShowPen         (void);
pascal void GetPen          (Point *pt);
pascal void GetPenState     (PenState *pnState);
pascal void SetPenState     (const PenState *pnState);
pascal void PenSize         (short width, short height);
pascal void PenMode         (short mode);
pascal void PenPat          (ConstPatternParam pat);
pascal void PenNormal       (void);
```

Changing the Background Bit Pattern

```
pascal void BackPat         (ConstPatternParam pat);
```

Drawing Lines

```
pascal void MoveTo          (short h, short v);
pascal void Move            (short dh, short dv);
pascal void LineTo          (short h, short v);
pascal void Line            (short dh, short dv);
```

Creating and Managing Rectangles

```
pascal void SetRect         (Rect *r, short left, short top, short right,
                             short bottom);
pascal void OffsetRect      (Rect *r, short dh, short dv);
pascal void InsetRect       (Rect *r, short dh, short dv);
pascal Boolean SectRect     (const Rect *src1, const Rect *src2,
                             Rect *dstRect);
pascal void UnionRect       (const Rect *src1, const Rect *src2,
                             Rect *dstRect);
pascal Boolean PtInRect     (Point pt, const Rect *r);
pascal void Pt2Rect         (Point pt1, Point pt2, Rect *dstRect);
pascal void PtToAngle       (const Rect *r, Point pt, short *angle);
pascal Boolean EqualRect    (const Rect *rect1, const Rect *rect2);
pascal Boolean EmptyRect    (const Rect *r);
```

Drawing Rectangles

```
pascal void FrameRect       (const Rect *r);
pascal void PaintRect       (const Rect *r);
pascal void FillRect        (const Rect *r, ConstPatternParam pat);
pascal void EraseRect       (const Rect *r);
pascal void InvertRect      (const Rect *r);
```

Drawing Rounded Rectangles

```
pascal void FrameRoundRect  (const Rect *r, short ovalWidth,
                             short ovalHeight);
pascal void PaintRoundRect  (const Rect *r, short ovalWidth,
                             short ovalHeight);
pascal void FillRoundRect   (const Rect *r, short ovalWidth,
                             short ovalHeight, ConstPatternParam pat);
```

```
pascal void EraseRoundRect  (const Rect *r, short ovalWidth,
                             short ovalHeight);
pascal void InvertRoundRect
                            (const Rect *r, short ovalWidth,
                             short ovalHeight);
```

Drawing Ovals

```
pascal void FrameOval       (const Rect *r);
pascal void PaintOval       (const Rect *r);
pascal void FillOval        (const Rect *r, ConstPatternParam pat);
pascal void EraseOval       (const Rect *r);
pascal void InvertOval      (const Rect *r);
```

Drawing Arcs and Wedges

```
pascal void FrameArc        (const Rect *r, short startAngle,
                             short arcAngle);
pascal void PaintArc        (const Rect *r, short startAngle,
                             short arcAngle);
pascal void FillArc         (const Rect *r, short startAngle,
                             short arcAngle, ConstPatternParam pat);
pascal void EraseArc        (const Rect *r, short startAngle,
                             short arcAngle);
pascal void InvertArc       (const Rect *r, short startAngle,
                             short arcAngle);
```

Creating and Managing Polygons

```
pascal PolyHandle OpenPoly  (void);
pascal void ClosePoly       (void);
pascal void OffsetPoly      (PolyHandle poly, short dh, short dv);
pascal void KillPoly        (PolyHandle poly);
```

Drawing and Painting Polygons

```
pascal void FramePoly       (PolyHandle poly);
pascal void PaintPoly       (PolyHandle poly);
pascal void FillPoly        (PolyHandle poly, ConstPatternParam pat);
pascal void ErasePoly       (PolyHandle poly);
pascal void InvertPoly      (PolyHandle poly);
```

Creating and Managing Regions

```
pascal RgnHandle NewRgn      (void);
pascal void OpenRgn          (void);
pascal void CloseRgn         (RgnHandle dstRgn);
pascal void DisposeRgn       (RgnHandle rgn);
pascal void CopyRgn          (RgnHandle srcRgn, RgnHandle dstRgn);
pascal void SetEmptyRgn      (RgnHandle rgn);
pascal void SetRectRgn       (RgnHandle rgn, short left, short top,
                              short right, short bottom);
pascal void RectRgn          (RgnHandle rgn, const Rect *r);
pascal void OffsetRgn        (RgnHandle rgn, short dh, short dv);
pascal void InsetRgn         (RgnHandle rgn, short dh, short dv);
pascal void SectRgn          (RgnHandle srcRgnA, RgnHandle srcRgnB,
                              RgnHandle dstRgn);
pascal void UnionRgn         (RgnHandle srcRgnA, RgnHandle srcRgnB,
                              RgnHandle dstRgn);
pascal void DiffRgn          (RgnHandle srcRgnA, RgnHandle srcRgnB,
                              RgnHandle dstRgn);
pascal void XorRgn           (RgnHandle srcRgnA, RgnHandle srcRgnB, RgnHandle
                              dstRgn);
pascal Boolean PtInRgn       (Point pt, RgnHandle rgn);
pascal Boolean RectInRgn     (const Rect *r, RgnHandle rgn);
pascal Boolean EqualRgn      (RgnHandle rgnA, RgnHandle rgnB);
pascal Boolean EmptyRgn      (RgnHandle rgn);
```

Drawing Regions

```
pascal void FrameRgn         (RgnHandle rgn);
pascal void PaintRgn         (RgnHandle rgn);
pascal void FillRgn          (RgnHandle rgn, ConstPatternParam pat);
pascal void EraseRgn         (RgnHandle rgn);
pascal void InvertRgn        (RgnHandle rgn);
```

Scaling and Mapping Points, Rectangles, Polygons, and Regions

```
pascal void ScalePt          (Point *pt, const Rect *srcRect,
                              const Rect *dstRect);
pascal void MapPt            (Point *pt, const Rect *srcRect,
                              const Rect *dstRect);
pascal void MapRect          (Rect *r, const Rect *srcRect,
                              const Rect *dstRect);
```

```
pascal void MapRgn          (RgnHandle rgn, const Rect *srcRect,
                             const Rect *dstRect);
pascal void MapPoly         (PolyHandle poly, const Rect *srcRect,
                             const Rect *dstRect);
```

Calculating Black-and-White Fills

```
pascal void SeedFill        (const void *srcPtr, void *dstPtr,
                             short srcRow, short dstRow, short height,
                             short words, short seedH, short seedV);
pascal void CalcMask        (const void *srcPtr, void *dstPtr,
                             short srcRow, short dstRow, short height,
                             short words);
```

Copying Images

```
pascal void CopyBits        (const BitMap *srcBits,
                             const BitMap *dstBits, const Rect *srcRect,
                             const Rect *dstRect, short mode,
                             RgnHandle maskRgn);
pascal void CopyMask        (const BitMap *srcBits,
                             const BitMap *maskBits, const BitMap *dstBits,
                             const Rect *srcRect, const Rect *maskRect,
                             const Rect *dstRect);
pascal void CopyDeepMask    (const BitMap *srcBits, const BitMap *maskBits,
                             const BitMap *dstBits, const Rect *srcRect,
                             const Rect *maskRect, const Rect *dstRect,
                             short mode, RgnHandle maskRgn);
```

Drawing With the Eight-Color System

```
pascal void ForeColor       (long color);
pascal void BackColor       (long color);
pascal void ColorBit        (short whichBit);
```

Determining Whether QuickDraw Has Finished Drawing

```
pascal Boolean QDDone       (GrafPtr port);
```

Getting Pattern Resources

```
pascal PatHandle GetPattern
                            (short patternID);
pascal void GetIndPattern   (Pattern thePat, short patternListID,
                             short index);
```

Customizing QuickDraw Operations

```
pascal void SetStdProcs     (QDProcs *procs);
pascal void StdText         (short count, const void *textAddr,
                             Point numer, Point denom);
pascal void StdLine         (Point newPt);
pascal void StdRect         (GrafVerb verb, const Rect *r);
pascal void StdRRect        (GrafVerb verb, const Rect *r,
                             short ovalWidth, short ovalHeight);
pascal void StdOval         (GrafVerb verb, const Rect *r);
pascal void StdArc          (GrafVerb verb, const Rect *r,
                             short startAngle, short arcAngle);
pascal void StdPoly         (GrafVerb verb, PolyHandle poly);
pascal void StdRgn          (GrafVerb verb, RgnHandle rgn);
pascal void StdBits         (const BitMap *srcBits,
                             const Rect *srcRect, const Rect *dstRect,
                             short mode, RgnHandle maskRgn);
pascal void StdComment      (short kind, short dataSize, Handle dataHandle);
pascal short StdTxtMeas     (short byteCount, const void *textAddr,
                             Point *numer, Point *denom, FontInfo *info);
pascal void StdGetPic       (void *dataPtr, short byteCount);
pascal void StdPutPic       (const void *dataPtr, short byteCount);
```

Assembly-Language Summary

Data Structures

Polygon Data Structure

0	`polySize`	word	total bytes in this structure
2	`polyBBox`	8 bytes	bounding rectangle
10	`polyPoints`	variable	vertices, each consisting of a long (point)

PenState Data Structure

0	`psLoc`	long	pen location
4	`psSize`	long	pen size
8	`psMode`	word	pattern mode
10	`psPat`	8 bytes	pattern

QDProcs Data Structure

0	`textProc`	long	pointer to text-drawing routine
4	`lineProc`	long	pointer to line-drawing routine
8	`rectProc`	long	pointer to rectangle-drawing routine
12	`rRectProc`	long	pointer to rounded rectangle–drawing routine
16	`ovalProc`	long	pointer to oval-drawing routine
20	`arcProc`	long	pointer to arc/wedge-drawing routine
24	`polyProc`	long	pointer to polygon-drawing routine
28	`rgnProc`	long	pointer to region-drawing routine
32	`bitsProc`	long	pointer to bit transfer routine
36	`commentProc`	long	pointer to picture comment–processing routine
40	`txMeasProc`	long	pointer to text-width measurement routine
44	`getPicProc`	long	pointer to picture retrieval routine
48	`putPicProc`	long	pointer to picture-saving routine

Trap Macro Requiring Routine Selector

`_QDExtensions`

Selector	Routine
$00040013	`QDDone`

Global Variables

`black`	All-black pattern.
`dkGray`	75% gray pattern.
`gray`	50% gray pattern.
`ltGray`	25% gray pattern.
`white`	All-white pattern.

Color QuickDraw

Contents

About Color QuickDraw 4-4
RGB Colors 4-4
The Color Drawing Environment: Color Graphics Ports 4-5
Pixel Maps 4-9
Pixel Patterns 4-12
Color QuickDraw's Translation of RGB Colors to Pixel Values 4-13
Colors on Grayscale Screens 4-17
Using Color QuickDraw 4-18
Initializing Color QuickDraw 4-19
Creating Color Graphics Ports 4-20
Drawing With Different Foreground Colors 4-21
Drawing With Pixel Patterns 4-23
Copying Pixels Between Color Graphics Ports 4-26
Boolean Transfer Modes With Color Pixels 4-32
Dithering 4-37
Arithmetic Transfer Modes 4-38
Highlighting 4-41
Color QuickDraw Reference 4-44
Data Structures 4-45
Color QuickDraw Routines 4-63
Opening and Closing Color Graphics Ports 4-63
Managing a Color Graphics Pen 4-67
Changing the Background Pixel Pattern 4-68
Drawing With Color QuickDraw Colors 4-70
Determining Current Colors and Best Intermediate Colors 4-79
Calculating Color Fills 4-82
Creating, Setting, and Disposing of Pixel Maps 4-85
Creating and Disposing of Pixel Patterns 4-87
Creating and Disposing of Color Tables 4-91
Retrieving Color QuickDraw Result Codes 4-94

Customizing Color QuickDraw Operations 4-96
Reporting Data Structure Changes to QuickDraw 4-97
Application-Defined Routine 4-101
Resources 4-102
The Pixel Pattern Resource 4-103
The Color Table Resource 4-104
The Color Icon Resource 4-105
Summary of Color QuickDraw 4-107
Pascal Summary 4-107
Constants 4-107
Data Types 4-109
Color QuickDraw Routines 4-113
Application-Defined Routine 4-115
C Summary 4-115
Constants 4-115
Data Types 4-118
Color QuickDraw Functions 4-122
Application-Defined Function 4-124
Assembly-Language Summary 4-124
Data Structures 4-124
Result Codes 4-128

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

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

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

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

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

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

About Color QuickDraw

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

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

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

RGB Colors

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

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

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

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

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

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

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

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

The Color Drawing Environment: Color Graphics Ports

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

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

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

Figure 4-1 The color graphics port

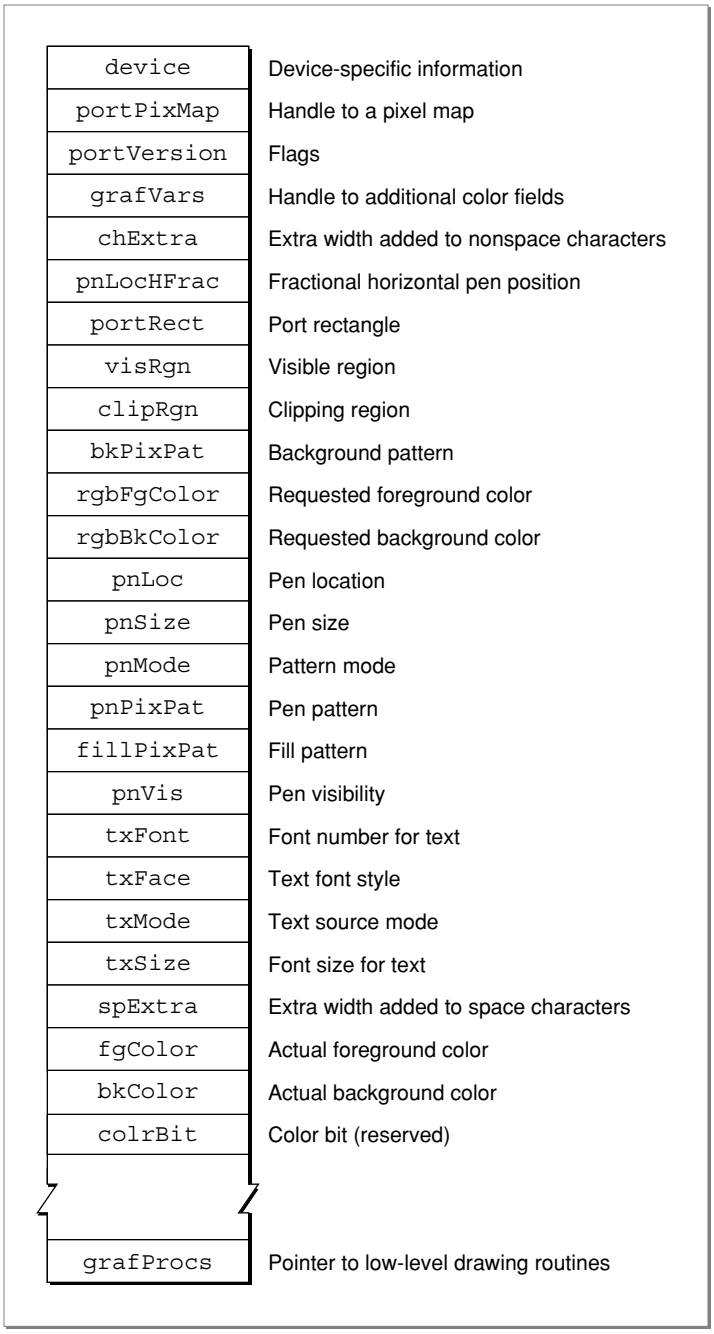


Table 4-3 on page 4-64 shows initial values for a `CGrafPort` record. A `CGrafPort` record is the same size as a `GrafPort` record (described in the chapter "Basic QuickDraw"), and most of the fields are identical for these two records. The important differences between these two data types are listed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

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

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

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

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

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

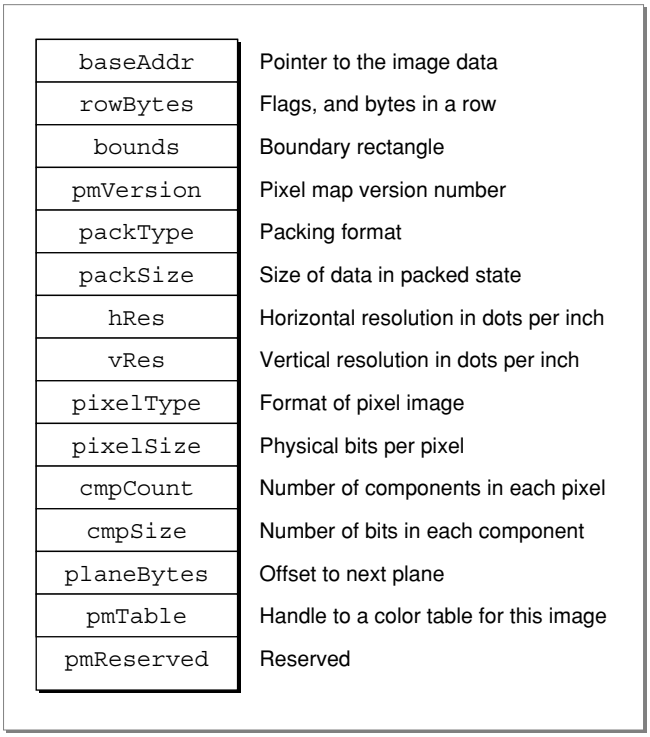
While the `CGrafPort` record contains information for a color window, there can be many windows on a screen, and even more than one screen. The `GDevice` record, described in the chapter "Graphics Devices," is the data structure that holds state information about a graphics device—such as the size of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

Pixel Maps

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

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

Figure 4-2 The pixel map



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Pixel Patterns

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

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

You can use the `FillCRect`, `FillCRoundRect`, `FillCArc`, `FillCPoly`, and `FillCRgn` procedures (described later in this chapter) to draw shapes with a pixel pattern other than the one specified in the `pnPixPat` field. When your application uses one of these procedures, the procedure stores the pattern your application specifies in the `fillPixPat` field of the `CGrafPort` record and then calls a low-level drawing routine that gets the pattern from that field.

Each graphics port also has a background pattern that's used when an area is erased (for example, by the `EraseRect`, `EraseRoundRect`, `EraseArc`, `ErasePoly`, and `EraseRgn` procedures, described in the chapter "QuickDraw Drawing") and when pixels are scrolled out of an area by the `ScrollRect` procedure, described in the chapter "Basic QuickDraw." Every color graphics port stores a background pixel pattern in the `bkPixPat` field of its `CGrafPort` record. Initially, every graphics port is assigned an all-white background pattern, but you can use the `BackPixPat` procedure to assign a different pixel pattern.

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

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

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

Color QuickDraw's Translation of RGB Colors to Pixel Values

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

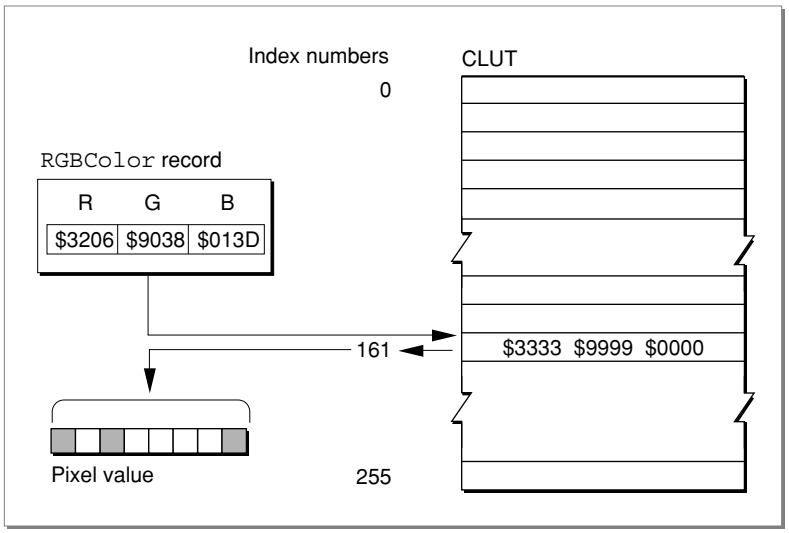
Your application never needs to handle pixel values. However, to clarify the relation between your application's 48-bit `RGBColor` records and the pixels that are actually displayed, this section presents some examples of how Color QuickDraw derives pixel values from your `RGBColor` records.

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

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

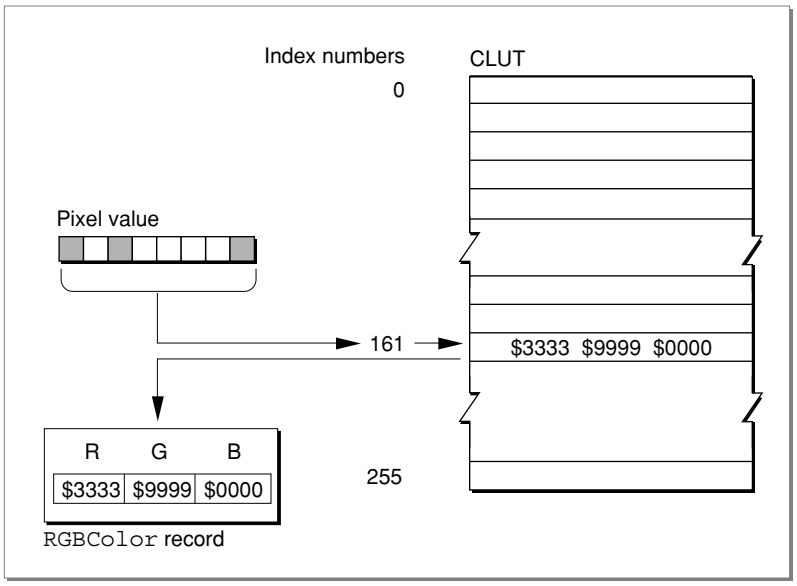
For example, the `RGBColor` record for a medium green pixel is represented on the left side of Figure 4-3. An application might create such a record and pass it to the `RGBForeColor` procedure, which sets the foreground color for drawing. In system software's standard 8-bit color lookup table (which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

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



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

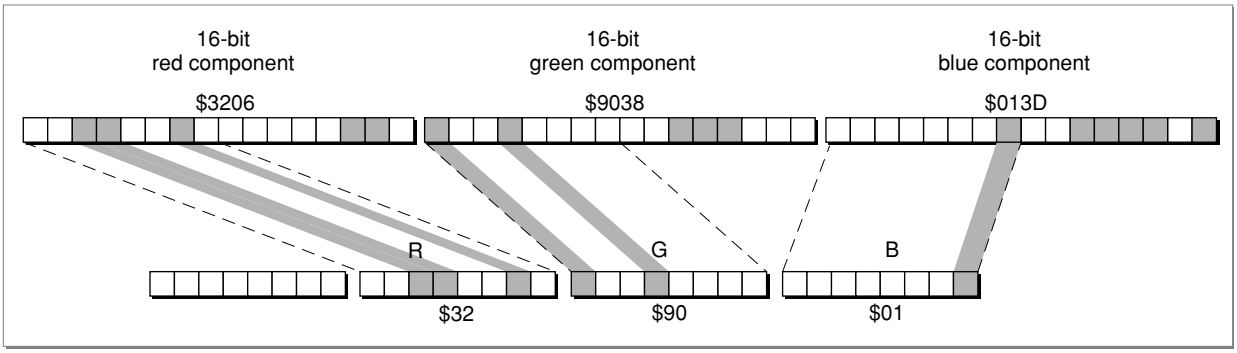
Figure 4-4 Translating an 8-bit pixel value on an indexed device to a 48-bit `RGBColor` record



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

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

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



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

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

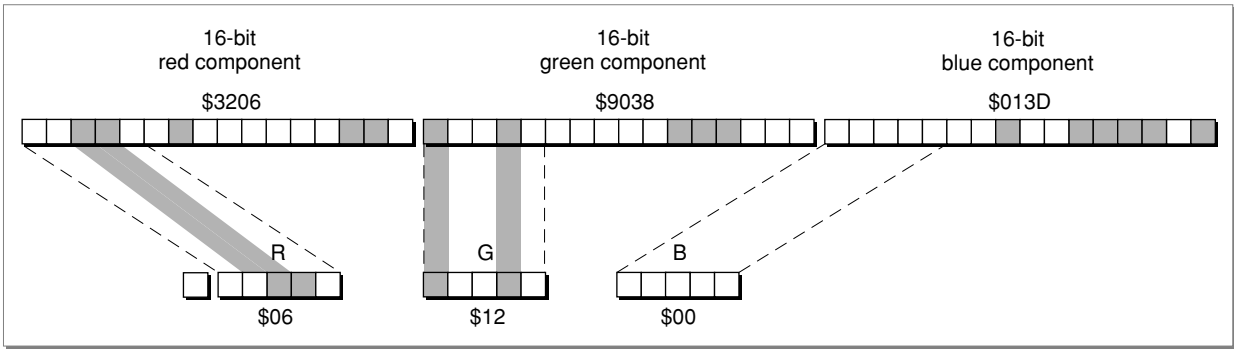


Figure 4-7 shows how Color QuickDraw expands a 32-bit pixel value to a 48-bit `RGBColor` record by dropping the unused high byte of the pixel value and doubling each of its 8-bit components. Note that the resulting 48-bit value differs in the least significant 8 bits of each component from the original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

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

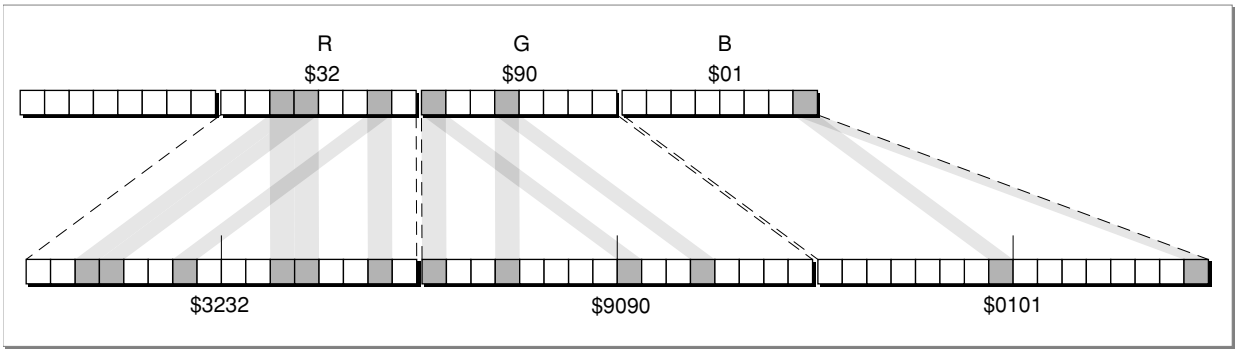
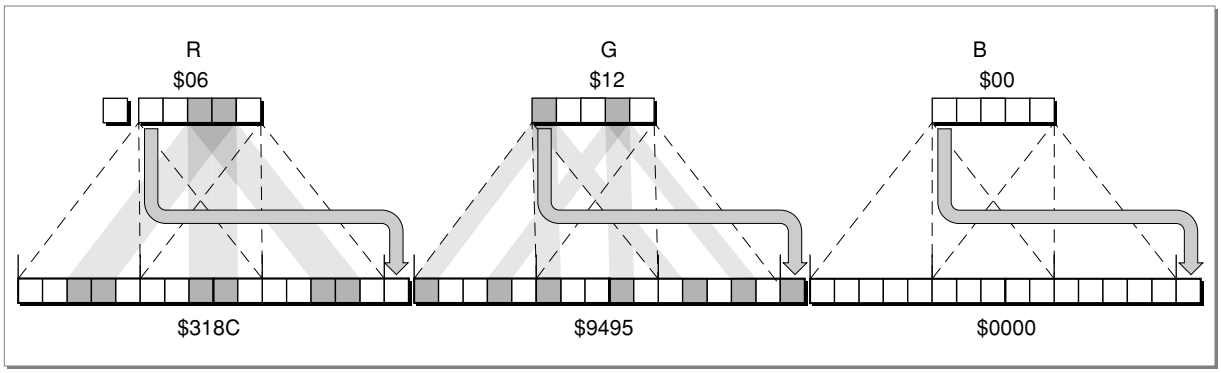


Figure 4-8 shows how Color QuickDraw expands a 16-bit pixel value to a 48-bit `RGBColor` record by dropping the unused high bit of the pixel value and inserting three copies of each 5-bit component and a copy of the most significant bit into each 16-bit field of the `RGBColor` record. Note that the result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

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



Colors on Grayscale Screens

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

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

Using Color QuickDraw

To use Color QuickDraw, you generally

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

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

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

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

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

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

Initializing Color QuickDraw

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

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

Creating Color Graphics Ports

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

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

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

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

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

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

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

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

Drawing With Different Foreground Colors

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

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

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

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

Listing 4-2 Changing the foreground color

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

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

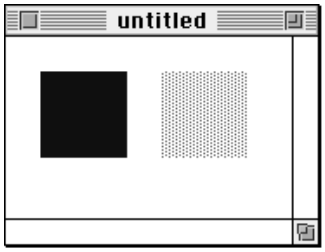
After using `SetRect` to create a rectangle, Listing 4-2 calls `PaintRect` to paint the rectangle. By default, the foreground color is black; by changing the foreground color to dark blue, every pixel that would normally be painted in black is instead painted in dark blue.

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

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

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

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



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

Drawing With Pixel Patterns

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

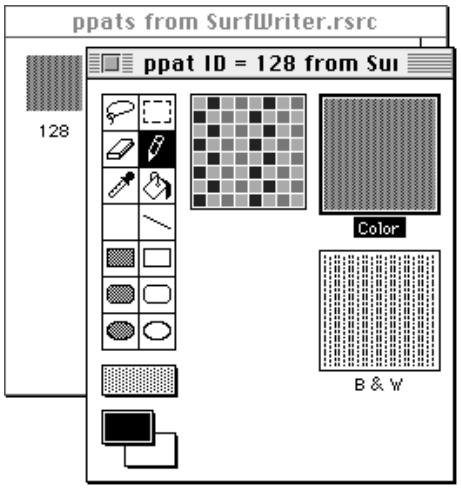
To set the pixel pattern to be used by the graphics pen in the current color graphics port, you use the `PenPixPat` procedure. To assign a pixel pattern as the background pattern, you use the `BackPixPat` procedure; this causes the `ScrollRect` procedure and the shape-erasing procedures (for example,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

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

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

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

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



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

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

Listing 4-3 Rez input for a pixel pattern resource

```
resource 'ppat' (128) {
   $"0001 0000 001C 0000 004E 0000 0000 FFFF"
   $"0000 0000 8292 1082 9210 8292 0000 0000"
   $"8002 0000 0000 0008 0008 0000 0000 0000"
   $"0000 0048 0000 0048 0000 0000 0002 0001"
   $"0002 0000 0000 0000 005E 0000 0000 1212"
   $"4848 1212 4848 1212 4848 1212 4848 0000"
   $"0000 0000 0002 0000 AAAA AAAA AAAA 0001"
   $"2222 2222 2222 0002 7777 7777 7777"
};
```

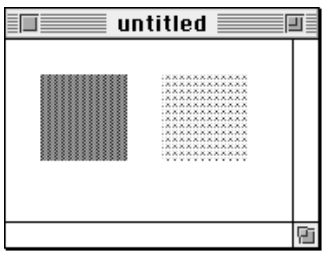
To retrieve the pixel pattern stored in a `'ppat'` resource, you can use the `GetPixPat` function. Listing 4-4 uses `GetPixPat` to retrieve the `'ppat'` resource created in Listing 4-3. To assign this pixel pattern to the graphics pen, Listing 4-4 uses the `PenPixPat` procedure.

Listing 4-4 Using pixel patterns to paint and fill

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

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

Figure 4-11 Painting and filling rectangles with pixel patterns



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

Copying Pixels Between Color Graphics Ports

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

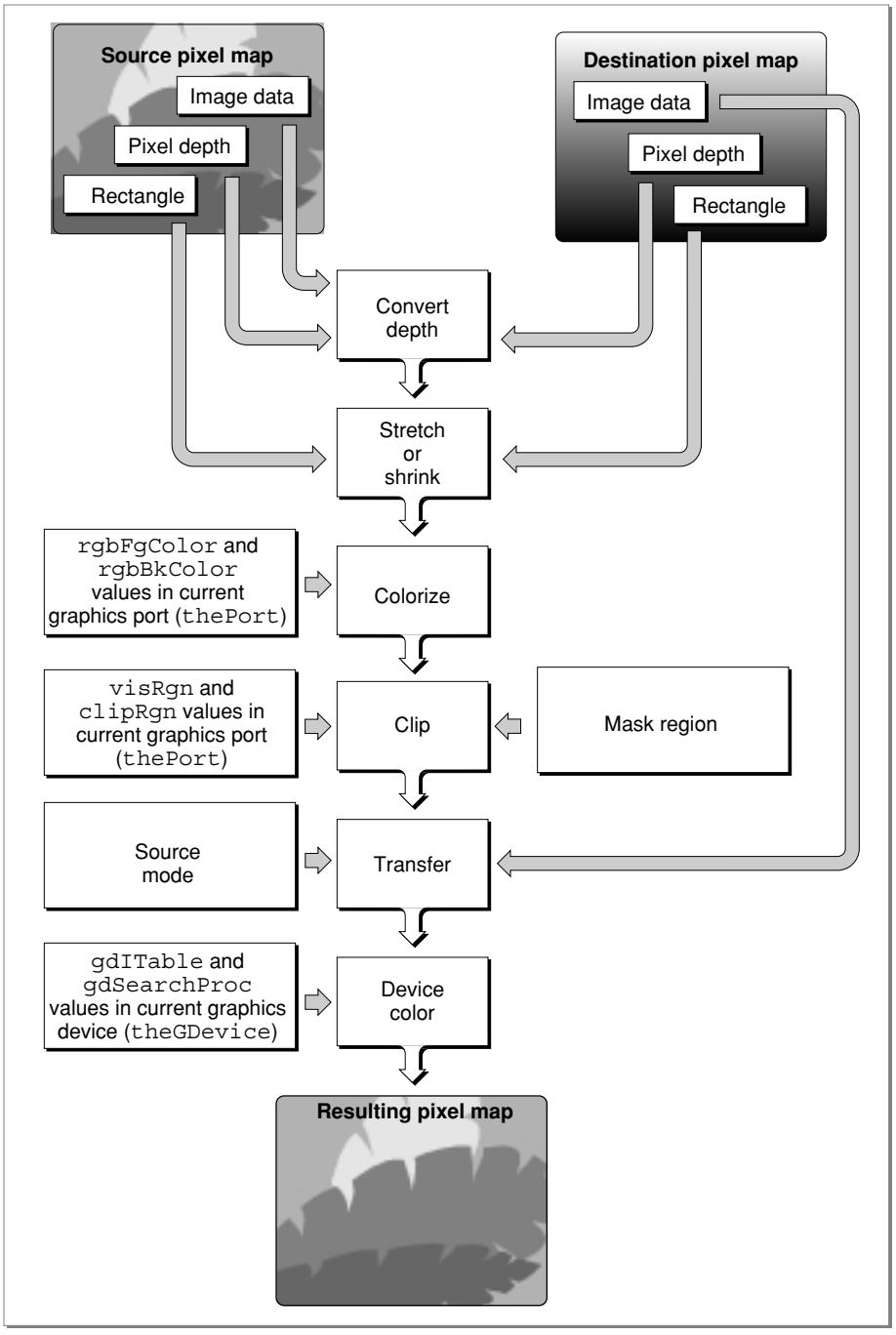
- The `CopyBits` procedure copies a pixel map or bitmap image to another graphics port, with facilities for resizing the image, modifying the image with transfer modes, and clipping the image to a region.
- The `CopyMask` procedure copies a pixel map or bitmap image to another graphics port, with facilities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

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

When using `CopyBits`, `CopyMask`, and `CopyDeepMask` to copy images between color graphics ports, you must coerce each port's `CGrafPtr` data type to a `GrafPtr` data type, dereference the `portBits` fields of each, and then pass these "bitmaps" in the `srcBits` and `dstBits` parameters. If your application copies a pixel image from a color graphics port called `MyColorPort`, in the `srcBits` parameter you could specify `GrafPtr(MyColorPort)^.portBits`. In a `CGrafPort` record, the high 2 bits of the `portVersion` field are set. This field, which shares the same position in a `CGrafPort` record as the `portBits.rowBytes` field in a `GrafPort` record, indicates to these routines that you have passed it a handle to a pixel map rather than a bitmap.

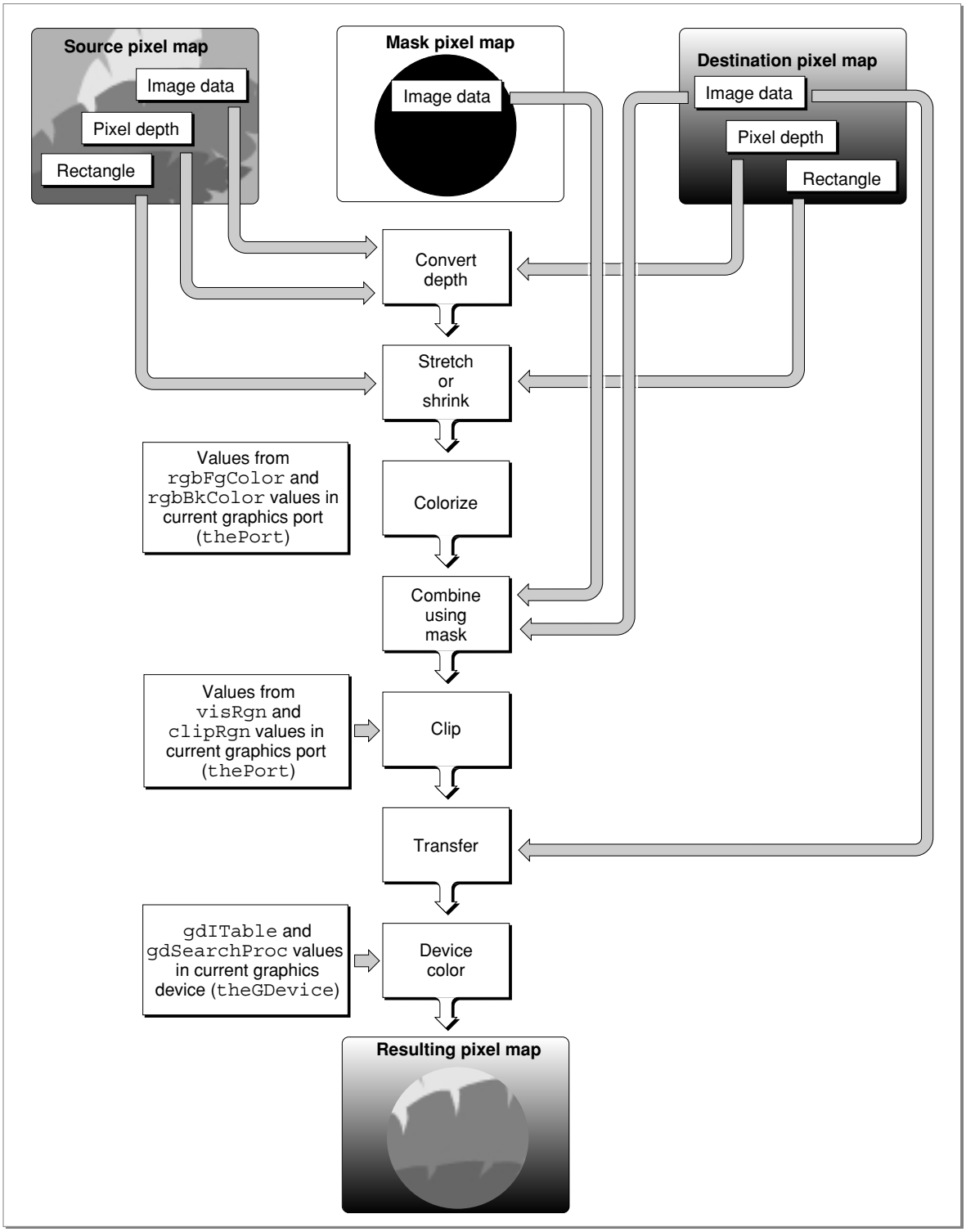
Color QuickDraw's processing sequence of the `CopyBits` procedure is illustrated in Figure 4-12. Listing 6-1 in the chapter "Offscreen Graphics Worlds" illustrates how to use `CopyBits` to transfer an image prepared in an offscreen graphics world to an onscreen color graphics port.

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



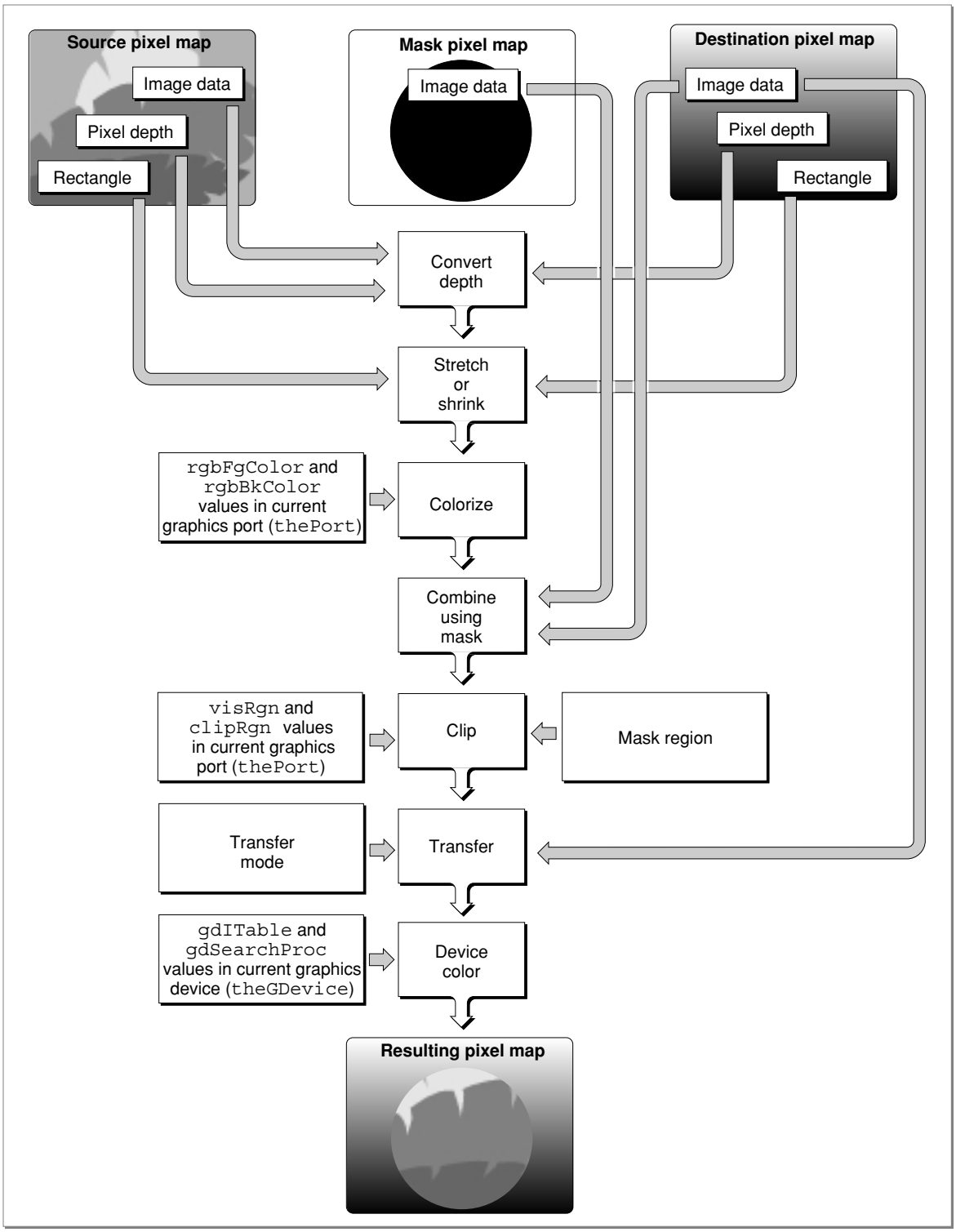
With the `CopyMask` procedure, you can supply a pixel map to act as a copying mask. The values of pixels in the mask act as weights that proportionally select between source and destination pixel values. The process is shown in Figure 4-13, and an example of the effect can be seen in Plate 3 at the front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

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



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

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



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

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

Boolean Transfer Modes With Color Pixels

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

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

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

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

Table 4-1 Boolean source modes with colored pixels

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Dithering

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

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

On computers running System 7, you can add dithering to any source mode by adding the following constant or the value it represents to the source mode:

```
CONST ditherCopy = 64;  {add to source mode for dithering}
```

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

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

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

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

Arithmetic Transfer Modes

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

```
CONST
   blend       = 32; {replace destination pixel with a blend }
                     { of the source and destination pixel }
                     { colors; if the destination is a bitmap or }
                     { 1-bit pixel map, revert to srcCopy mode}
   addPin      = 33; {replace destination pixel with the sum of }
                     { the source and destination pixel colors-- }
                     { up to a maximum allowable value; if }
                     { the destination is a bitmap or }
                     { 1-bit pixel map, revert to srcBic mode}
   addOver     = 34; {replace destination pixel with the sum of }
                     { the source and destination pixel colors-- }
                     { but if the value of the red, green, or }
                     { blue component exceeds 65,536, then }
                     { subtract 65,536 from that value; if the }
                     { destination is a bitmap or 1-bit }
                     { pixel map, revert to srcXor mode}
```

```
subPin      = 35; {replace destination pixel with the }
                  { difference of the source and destination }
                  { pixel colors--but not less than a minimum }
                  { allowable value; if the destination }
                  { is a bitmap or 1-bit pixel map, revert to }
                  { srcOr mode}
transparent = 36; {replace the destination pixel with the }
                  { source pixel if the source pixel isn't }
                  { equal to the background color}
addMax      = 37; {compare the source and destination pixels, }
                  { and replace the destination pixel with }
                  { the color containing the greater }
                  { saturation of each of the RGB components; }
                  { if the destination is a bitmap or }
                  { 1-bit pixel map, revert to srcBic mode}
subOver     = 38; {replace destination pixel with the }
                  { difference of the source and destination }
                  { pixel colors--but if the value of a red, }
                  { green, or blue component is }
                  { less than 0, add the negative result to }
                  { 65,536; if the destination is a bitmap or }
                  { 1-bit pixel map, revert to srcXor mode}
adMin       = 39; {compare the source and destination pixels, }
                  { and replace the destination pixel with }
                  { the color containing the lesser }
                  { saturation of each of the RGB components; }
                  { if the destination is a bitmap or }
                  { 1-bit pixel map, revert to srcOr mode}
```

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

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

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

Note
The arithmetic transfer modes have no coloration effects. ◆

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

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

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

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

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

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

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

```
transparent + patCopy
```

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

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

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

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

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

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

Highlighting

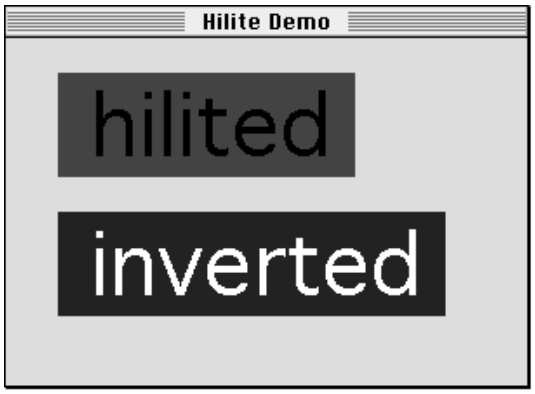
When **highlighting,** Color QuickDraw replaces the background color with the highlight color when your application draws or copies images between graphics ports. This has the visual effect of using a highlighting pen to select the object. For instance, TextEdit (described in *Inside Macintosh: Text*)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

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

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

within the rectangle's boundary. On a black-and-white monitor, this changes all black pixels in the shape to white, and changes all white pixels to black. Although this procedure operates on color pixels in color graphics ports, the results are predictable only with direct pixels or 1-bit pixel maps.

Figure 4-15 Difference between highlighting and inverting



The global variable `HiliteRGB` is read from parameter RAM when the machine starts. Basic graphics ports use the color stored in the `HiliteRGB` global variable as the highlight color. Color graphics ports default to the `HiliteRGB` global variable, but you can override this by using the `HiliteColor` procedure, described on page 4-78.

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

The following constant represents the highlight bit:

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

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

Listing 4-6 Setting the highlight bit

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

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

Listing 4-7 Using highlighting for text

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

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

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

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

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

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

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

Color QuickDraw Reference

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

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

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

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

Data Structures

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

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

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

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

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

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

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

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

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

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

PixMap

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

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

Field descriptions

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

▲ **WARNING**

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

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

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

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

CGrafPort

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

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

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

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

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

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

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

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

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

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

Field descriptions

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

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

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

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

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

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

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

`pnLoc`	The point where QuickDraw will begin drawing the next line, shape, or character. It can be anywhere on the coordinate plane; there are no restrictions on the movement or placement of the pen. The location of the graphics pen is a point in the graphics port's coordinate system, not a pixel in a pixel image. The upper-left corner of the pen is at the pen location; the graphics pen hangs below and to the right of this point. The graphics pen is described in detail in the chapter "QuickDraw Drawing."
`pnSize`	The vertical height and horizontal width of the graphics pen. The default size is a 1-by-1 pixel square; the vertical height and horizontal width can range from 0 by 0 to 32,767 by 32,767. If either the pen width or the pen height is 0, the pen does not draw. Heights or widths of less than 0 are undefined. You can use the `PenSize` procedure (described in the chapter "QuickDraw Drawing") to change the value in this field.
`pnMode`	The pattern mode—that is, a Boolean operation that determines the how Color QuickDraw transfers the pen pattern to the pixel map during drawing operations. When the graphics pen draws into a pixel map, Color QuickDraw first determines what pixels in the pixel image are affected and finds their corresponding pixels in the pen pattern. Color QuickDraw then does a pixel-by-pixel comparison based on the pattern mode, which specifies one of eight Boolean transfer operations to perform. Color QuickDraw stores the resulting pixel in its proper place in the image. Pattern modes for a color graphics port are described in "Boolean Transfer Modes With Color Pixels" beginning on page 4-32.
`pnPixPat`	A handle to a `PixPat` record (described on page 4-58) that describes a pixel pattern used like the ink in the graphics pen. Color QuickDraw uses the pixel pattern defined in the `PixPat` record when you use the `Line` and `LineTo` procedures to draw lines with the pen, framing procedures such as `FrameRect` to draw shape outlines with the pen, and painting procedures such as `PaintRect` to paint shapes with the pen.
`fillPixPat`	A handle to a `PixPat` record (described on page 4-58) that describes the pixel pattern that's used when you call a procedure such as `FillRect` to fill an area. Notice that this is not in the same location as the `fillPat` field in a `GrafPort` record.
`pnVis`	The graphics pen's visibility—that is, whether it draws on the screen. The graphics pen is described in detail in the chapter "QuickDraw Drawing."

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

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

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

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

RGBColor

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

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

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

Field descriptions

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

ColorSpec

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

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

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

Field descriptions

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

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

ColorTable

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

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

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

Field descriptions

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

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

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

MatchRec

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

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

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

Field descriptions

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

PixPat

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

A `PixPat` record is defined as follows:

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

Field descriptions

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

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

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

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

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

`patXMap` Reserved for use by Color QuickDraw.

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

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

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

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

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

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

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

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

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

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

CQDProcs

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

```
CQDProcsPtr = ^CQDProcs
CQDProcs    =
RECORD
   textProc:      Ptr;  {text drawing}
   lineProc:      Ptr;  {line drawing}
   rectProc:      Ptr;  {rectangle drawing}
   rRectProc:     Ptr;  {roundRect drawing}
   ovalProc:      Ptr;  {oval drawing}
   arcProc:       Ptr;  {arc/wedge drawing}
   polyProc:      Ptr;  {polygon drawing}
   rgnProc:       Ptr;  {region drawing}
   bitsProc:      Ptr;  {bit transfer}
   commentProc:   Ptr;  {picture comment processing}
   txMeasProc:    Ptr;  {text width measurement}
   getPicProc:    Ptr;  {picture retrieval}
   putPicProc:    Ptr;  {picture saving}
   opcodeProc:    Ptr;  {reserved for future use}
   newProc1:      Ptr;  {reserved for future use}
   newProc2:      Ptr;  {reserved for future use}
```

```
   newProc3:      Ptr;  {reserved for future use}
   newProc4:      Ptr;  {reserved for future use}
   newProc5:      Ptr;  {reserved for future use}
   newProc6:      Ptr;  {reserved for future use}
END;
```

Field descriptions

`textProc`	A pointer to the low-level routine that draws text. The standard QuickDraw routine is the `StdText` procedure.
`lineProc`	A pointer to the low-level routine that draws lines. The standard QuickDraw routine is the `StdLine` procedure.
`rectProc`	A pointer to the low-level routine that draws rectangles. The standard QuickDraw routine is the `StdRect` procedure.
`rRectProc`	A pointer to the low-level routine that draws rounded rectangles. The standard QuickDraw routine is the `StdRRect` procedure.
`ovalProc`	A pointer to the low-level routine that draws ovals. The standard QuickDraw routine is the `StdOval` procedure.
`arcProc`	A pointer to the low-level routine that draws arcs. The standard QuickDraw routine is the `StdArc` procedure.
`polyProc`	A pointer to the low-level routine that draws polygons. The standard QuickDraw routine is the `StdPoly` procedure.
`rgnProc`	A pointer to the low-level routine that draws regions. The standard QuickDraw routine is the `StdRgn` procedure.
`bitsProc`	A pointer to the low-level routine that copies bitmaps. The standard QuickDraw routine is the `StdBits` procedure.
`commentProc`	A pointer to the low-level routine for processing a picture comment. The standard QuickDraw routine is the `StdComment` procedure.
`txMeasProc`	A pointer to the low-level routine for measuring text width. The standard QuickDraw routine is the `StdTxMeas` function.
`getPicProc`	A pointer to the low-level routine for retrieving information from the definition of a picture. The standard QuickDraw routine is the `StdGetPic` procedure.
`putPicProc`	A pointer to the low-level routine for saving information as the definition of a picture. The standard QuickDraw routine is the `StdPutPic` procedure.
`opcodeProc`	Reserved for future use.
`newProc1`	Reserved for future use.
`newProc2`	Reserved for future use.
`newProc3`	Reserved for future use.
`newProc4`	Reserved for future use.
`newProc5`	Reserved for future use.
`newProc6`	Reserved for future use.

GrafVars

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

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

Field descriptions

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

`rgbHiliteColor`
The highlight color for this graphics port.

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

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

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

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

`pmFlags` Flags private to the Palette Manager.

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

Color QuickDraw Routines

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

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

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

Opening and Closing Color Graphics Ports

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

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

OpenCPort

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

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

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

DESCRIPTION

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

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

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

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

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

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

SPECIAL CONSIDERATIONS

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

InitCPort

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

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

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

DESCRIPTION

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

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

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

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

SPECIAL CONSIDERATIONS

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

SEE ALSO

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

CloseCPort

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

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

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

DESCRIPTION

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

SPECIAL CONSIDERATIONS

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

Managing a Color Graphics Pen

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

PenPixPat

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

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

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

DESCRIPTION

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

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

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

SPECIAL CONSIDERATIONS

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

SEE ALSO

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

The `GrafPort` record is described in the chapter "Basic QuickDraw." To set the graphics pen to use a bit pattern, you can also use the `PenPat` procedure, which is described in the chapter "QuickDraw Drawing" in this book. The `PenPat` procedure creates a handle, of type `PixPatHandle`, for the bit pattern and stores this handle in the `pnPixPat` field of the `CGrafPort` record.

Changing the Background Pixel Pattern

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

BackPixPat

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

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

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

DESCRIPTION

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

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

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

SPECIAL CONSIDERATIONS

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

SEE ALSO

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

The `GrafPort` record is described in the chapter "Basic QuickDraw." To set the background pattern to a bit pattern, you can also use the `BackPat` procedure, which is described in the chapter "QuickDraw Drawing" in this book. The `BackPat` procedure creates a handle, of type `PixPatHandle`, for the bit pattern and stores this handle in the `bkPixPat` field of the `CGrafPort` record. As in basic graphics ports, Color QuickDraw draws patterns in color graphics ports at the time of drawing, not at the time you use `BackPat` to set the pattern.

Drawing With Color QuickDraw Colors

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

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

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

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

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

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

RGBForeColor

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

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

`color` An `RGBColor` record.

DESCRIPTION

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

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

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

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

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

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

SPECIAL CONSIDERATIONS

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

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

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

SEE ALSO

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

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

RGBBackColor

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

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

`color` An `RGBColor` record.

DESCRIPTION

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

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

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

SPECIAL CONSIDERATIONS

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

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

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

SEE ALSO

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

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

SetCPixel

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

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

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

DESCRIPTION

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

SPECIAL CONSIDERATIONS

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

SEE ALSO

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

FillCRect

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

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

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

DESCRIPTION

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

SPECIAL CONSIDERATIONS

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

FillCRoundRect

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

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

`r`	The rectangle that defines the rounded rectangle's boundaries.
`ovalWidth`	The width of the oval defining the rounded corner.
`ovalHeight`	The height of the oval defining the rounded corner.
`ppat`	A handle to the `PixPat` record for the pixel pattern to be used for the fill.

DESCRIPTION

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

SPECIAL CONSIDERATIONS

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

FillCOval

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

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

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

DESCRIPTION

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

SPECIAL CONSIDERATIONS

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

FillCArc

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

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

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

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

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

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

DESCRIPTION

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

SPECIAL CONSIDERATIONS

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

FillCPoly

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

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

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

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

DESCRIPTION

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

SPECIAL CONSIDERATIONS

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

FillCRgn

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

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

`rgn` A handle to the region to be filled.

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

DESCRIPTION

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

SPECIAL CONSIDERATIONS

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

OpColor

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

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

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

DESCRIPTION

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

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

SEE ALSO

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

HiliteColor

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

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

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

DESCRIPTION

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

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

SEE ALSO

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

Determining Current Colors and Best Intermediate Colors

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

GetForeColor

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

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

`color` An `RGBColor` record.

DESCRIPTION

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

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

SPECIAL CONSIDERATIONS

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

SEE ALSO

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

GetBackColor

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

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

`color` An `RGBColor` record.

DESCRIPTION

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

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

SPECIAL CONSIDERATIONS

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

SEE ALSO

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

GetCPixel

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

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

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

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

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

DESCRIPTION

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

SEE ALSO

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

GetGray

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

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

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

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

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

DESCRIPTION

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

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

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

Calculating Color Fills

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

SeedCFill

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

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

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

DESCRIPTION

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

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

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

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

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

SEE ALSO

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

CalcCMask

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

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

`srcBits`	The source image. If the image is in a pixel map, you must coerce its `PixMap` record to a `BitMap` record.
`dstBits`	The destination image, a `BitMap` record.
`srcRect`	The rectangle of the source image.
`dstRect`	The rectangle of the destination image.
`seedRGB`	An `RGBColor` record specifying the color for pixels that should not be filled.
`matchProc`	An optional matching procedure.
`matchData`	Data for the optional matching procedure.

DESCRIPTION

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

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

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

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

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

SEE ALSO

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

Creating, Setting, and Disposing of Pixel Maps

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

NewPixMap

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

```
FUNCTION NewPixMap: PixMapHandle;
```

DESCRIPTION

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

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

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

SPECIAL CONSIDERATIONS

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

CopyPixMap

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

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

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

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

DESCRIPTION

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

SetPortPix

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

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

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

DESCRIPTION

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

SPECIAL CONSIDERATIONS

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

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

DisposePixMap

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

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

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

DESCRIPTION

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

SPECIAL CONSIDERATIONS

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

Creating and Disposing of Pixel Patterns

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

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

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

GetPixPat

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

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

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

DESCRIPTION

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

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

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

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

SPECIAL CONSIDERATIONS

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

SEE ALSO

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

NewPixPat

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

```
FUNCTION NewPixPat: PixPatHandle;
```

DESCRIPTION

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

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

(width of bounds) × `pixelSize`/8

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

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

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

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

SPECIAL CONSIDERATIONS

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

CopyPixPat

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

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

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

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

DESCRIPTION

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

SEE ALSO

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

MakeRGBPat

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

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

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

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

DESCRIPTION

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

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

SPECIAL CONSIDERATIONS

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

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

DisposePixPat

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

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

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

DESCRIPTION

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

Creating and Disposing of Color Tables

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

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

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

GetCTable

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

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

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

DESCRIPTION

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

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

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

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

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

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

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

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

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

SPECIAL CONSIDERATIONS

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

SEE ALSO

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

DisposeCTable

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

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

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

DESCRIPTION

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

Retrieving Color QuickDraw Result Codes

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

QDError

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

```
FUNCTION QDError: Integer;
```

DESCRIPTION

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

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

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

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

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

RESULT CODES

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

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

SECIAL CONSIDERATIONS

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

SEE ALSO

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

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

Customizing Color QuickDraw Operations

For each shape that QuickDraw can draw, there are procedures that perform these basic graphics operations on the shape: framing, painting, erasing, inverting, and filling. As described in the chapter "QuickDraw Drawing" in this book, those procedures in turn call a low-level drawing routine for the shape. For example, the `FrameOval`, `PaintOval`, `EraseOval`, `InvertOval`, and `FillOval` procedures all call the low-level procedure `StdOval`, which draws the oval.

The `grafProcs` field of a `CGrafPort` record determines which low-level routines are called. If that field contains the value of `NIL`, the standard routines are called. You can set the `grafProcs` field to point to a record of pointers to your own routines. This record of pointers is defined by a data structure of type `CQDProcs`. By changing these pointers, you can install your own routines, and either completely override the standard ones or call them after your routines have modified their parameters as necessary.

To assist you in setting up a record, QuickDraw provides the `SetStdCProcs` procedure. You can use the `SetStdCProcs` procedure to set all the fields of the `CQDProcs` record to point to the standard routines. You can then reset the ones with which you are concerned.

SetStdCProcs

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

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

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

DESCRIPTION

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

SPECIAL CONSIDERATIONS

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

SEE ALSO

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

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

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

Reporting Data Structure Changes to QuickDraw

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

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

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

CTabChanged

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

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

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

DESCRIPTION

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

SPECIAL CONSIDERATIONS

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

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

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
`_QDExtensions`	$00040007

SEE ALSO

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

PixPatChanged

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

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

`ppat` A handle to the changed pixel pattern.

DESCRIPTION

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

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

SPECIAL CONSIDERATIONS

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

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

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
`_QDExtensions`	$00040008

PortChanged

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

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

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

DESCRIPTION

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

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

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

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
`_QDExtensions`	$00040009

GDeviceChanged

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

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

DESCRIPTION

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

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

SPECIAL CONSIDERATIONS

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

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

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
`_QDExtensions`	$0004000A

Application-Defined Routine

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

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

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

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

MyColorSearch

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

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

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

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

DESCRIPTION

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

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

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

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

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

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

Resources

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

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

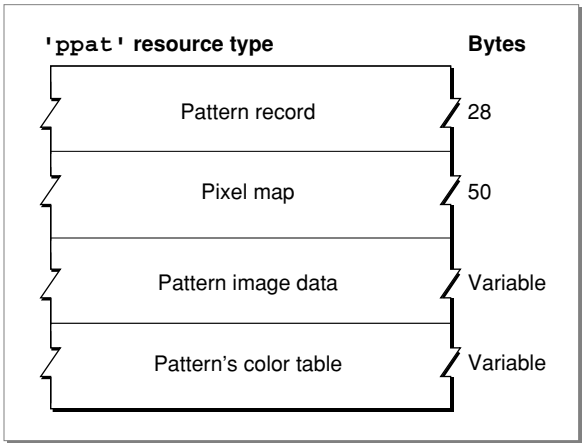
The Pixel Pattern Resource

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

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

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

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



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

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

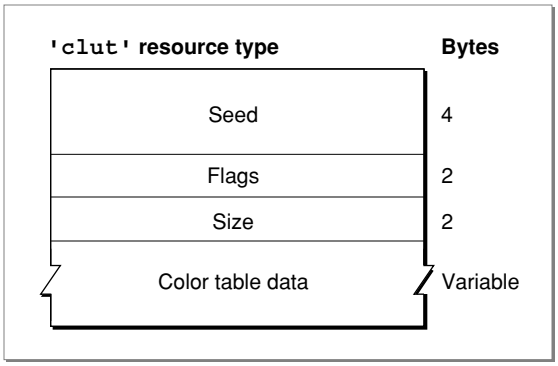
The Color Table Resource

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

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

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

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



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

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

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

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

The Color Icon Resource

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

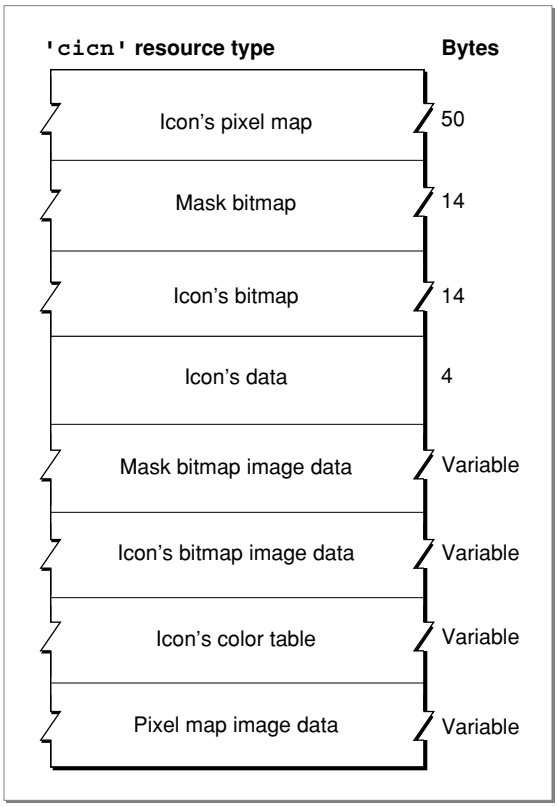
Using color icon resources, you can create icons similar to the ones that the Finder uses to display your application's files on the desktop; however, the Finder does *not* use or display any resources that you create of type `'cicn'`. Instead, your application uses icon resources of type `'cicn'` to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

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

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

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

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



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

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

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

Summary of Color QuickDraw

Pascal Summary

Constants

```
CONST
   {checking for Color QuickDraw and its features}
   gestaltQuickdrawVersion = 'qd  ';   {Gestalt selector for Color QuickDraw}
   gestalt8BitQD     = $100;  {8-bit Color QD}
   gestalt32BitQD    = $200;  {32-bit Color QD}
   gestalt32BitQD11  = $210;  {32-bit Color QDv1.1}
   gestalt32BitQD12  = $220;  {32-bit Color QDv1.2}
   gestalt32BitQD13  = $230;  {System 7: 32-bit Color QDv1.3}
   gestaltQuickdrawFeatures   = 'qdrw';   {Gestalt selector for Color }
                                          { QuickDraw features}
   gestaltHasColor         = 0;  {Color QuickDraw is present}
   gestaltHasDeepGWorlds   = 1;  {GWorlds deeper than 1 bit}
   gestaltHasDirectPixMaps = 2;  {PixMaps can be direct--16 or 32 bit}
   gestaltHasGrayishTextOr = 3;  {supports text mode grayishTextOr}
                  {source modes for color graphics ports}
   srcCopy     = 0;  {determine how close the color of the source pixel is }
                     { to black, and assign this relative amount of }
                     { foreground color to the destination pixel; determine }
                     { how close the color of the source pixel is to white, }
                     { and assign this relative amount of background }
                     { color to the destination pixel}
   srcOr       = 1;  {determine how close the color of the source pixel is }
                     { to black, and assign this relative amount of }
                     { foreground color to the destination pixel}
   srcXor      = 2;  {where source pixel is black, invert the destination }
                     { pixel--for a colored destination pixel, use the }
                     { complement of its color if the pixel is direct, }
                     { invert its index if the pixel is indexed}
   srcBic      = 3;  {determine how close the color of the source pixel is }
                     { to black, and assign this relative amount of }
                     { background color to the destination pixel}
```

```
notSrcCopy  = 4;  {determine how close the color of the source pixel is }
                  { to black, and assign this relative amount of }
                  { background color to the destination pixel; determine }
                  { how close the color of the source pixel is to white, }
                  { and assign this relative amount of foreground color }
                  { to the destination pixel}
notSrcOr    = 5;  {determine how close the color of the source pixel is }
                  { to white, and assign this relative amount of }
                  { foreground color to the destination pixel}
notSrcXor   = 6;  {where source pixel is white, invert the destination }
                  { pixel--for a colored destination pixel, use the }
                  { complement of its color if the pixel is direct, }
                  { invert its index if the pixel is indexed}
notSrcBic   = 7;  {determine how close the color of the source pixel is }
                  { to white, and assign this relative amount of }
                  { background color to the destination pixel}

{special text transfer mode}
grayishTextOr = 49;

{arithmetic transfer modes available in Color QuickDraw}
blend       = 32; {replace destination pixel with a blend of the source }
                  { and destination pixel colors; if the destination is }
                  { a bitmap or 1-bit pixel map, revert to srcCopy mode}
addPin      = 33; {replace destination pixel with the sum of the source }
                  { and destination pixel colors--up to a maximum }
                  { allowable value; if the destination is a bitmap or }
                  { 1-bit pixel map, revert to srcBic mode}
addOver     = 34; {replace destination pixel with the sum of the source }
                  { and destination pixel colors--but if the value of }
                  { the red, green, or blue component exceeds 65,536, }
                  { then subtract 65,536 from that value; if the }
                  { destination is a bitmap or 1-bit pixel map, revert }
                  { to srcXor mode}
subPin      = 35; {replace destination pixel with the difference of the }
                  { source and destination pixel colors--but not less }
                  { than a minimum allowable value; if the destination }
                  { is a bitmap or 1-bit pixel map, revert to srcOr mode}
addMax      = 37; {compare the source and destination pixels, and }
                  { replace the destination pixel with the color }
                  { containing the greater saturation of each of the RGB }
                  { components; if the destination is a bitmap or }
                  { 1-bit pixel map, revert to srcBic mode}
```

```
   subOver     = 38; {replace destination pixel with the difference of the }
                     { source and destination pixel colors--but if the }
                     { value of the red, green, or blue component is less }
                     { than 0, add the negative result to 65,536; if the }
                     { destination is a bitmap or 1-bit pixel map, revert }
                     { to srcXor mode}
   adMin       = 39; {compare the source and destination pixels, and }
                     { replace the destination pixel with the color }
                     { containing the lesser saturation of each of the RGB }
                     { components; if the destination is a bitmap or }
                     { 1-bit pixel map, revert to srcOr mode}

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

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

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

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

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

Data Types

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

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

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

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

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

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

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

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

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

CQDProcsPtr       = ^CQDProcs
CQDProcs          =
RECORD
   textProc:      Ptr;  {text drawing}
   lineProc:      Ptr;  {line drawing}
   rectProc:      Ptr;  {rectangle drawing}
   rRectProc:     Ptr;  {rounded rectangle drawing}
   ovalProc:      Ptr;  {oval drawing}
   arcProc:       Ptr;  {arc and wedge drawing}
   polyProc:      Ptr;  {polygon drawing}
   rgnProc:       Ptr;  {region drawing}
   bitsProc:      Ptr;  {bit transfer}
   commentProc:   Ptr;  {picture comment processing}
   txMeasProc:    Ptr;  {text width measurement}
   getPicProc:    Ptr;  {picture retrieval}
   putPicProc:    Ptr;  {picture saving}
   opcodeProc:    Ptr;  {reserved for future use}
   newProc1:      Ptr;  {reserved for future use}
   newProc2:      Ptr;  {reserved for future use}
   newProc3:      Ptr;  {reserved for future use}
   newProc4:      Ptr;  {reserved for future use}
   newProc5:      Ptr;  {reserved for future use}
   newProc6:      Ptr;  {reserved for future use}
END;
```

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

Color QuickDraw Routines

Opening and Closing Color Graphics Ports

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

Managing a Color Graphics Pen

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

Changing the Background Pixel Pattern

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

Drawing With Color QuickDraw Colors

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

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

Determining Current Colors and Best Intermediate Colors

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

Calculating Color Fills

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

Creating, Setting, and Disposing of Pixel Maps

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

Creating and Disposing of Pixel Patterns

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

Creating and Disposing of Color Tables

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

Retrieving Color QuickDraw Result Codes

```
FUNCTION QDError:           Integer;
```

Customizing Color QuickDraw Operations

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

Reporting Data Structure Changes to QuickDraw

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

Application-Defined Routine

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

C Summary

Constants

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

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

/* source modes for color graphics ports */
srcCopy     = 0,  /* determine how close the color of the source pixel is
                     to black, and assign this relative amount of
                     foreground color to the destination pixel; determine
                     how close the color of the source pixel is to white,
                     and assign this relative amount of background color
                     to the destination pixel */
srcOr       = 1,  /* determine how close the color of the source pixel is
                     to black, and assign this relative amount of
                     foreground color to the destination pixel */
srcXor      = 2,  /* where source pixel is black, invert the destination
                     pixel--for a colored destination pixel, use the
                     complement of its color if the pixel is direct,
                     invert its index if the pixel is indexed */
srcBic      = 3,  /* determine how close the color of the source pixel is
                     to black, and assign this relative amount of
                     background color to the destination pixel */
notSrcCopy  = 4,  /* determine how close the color of the source pixel is
                     to black, and assign this relative amount of
                     background color to the destination pixel; determine
                     how close the color of the source pixel is to white,
                     and assign this relative amount of foreground color
                     to the destination pixel */
notSrcOr    = 5,  /* determine how close the color of the source pixel is
                     to white, and assign this relative amount of
                     foreground color to the destination pixel */
notSrcXor   = 6,  /* where source pixel is white, invert destination
                     pixel--for a colored destination pixel, use the
                     complement of its color if the pixel is direct,
                     invert its index if the pixel is indexed */
notSrcBic   = 7,  /* determine how close the color of the source pixel is
                     to white, and assign this relative amount of
                     background color to the destination pixel */

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

```
/* arithmetic transfer modes available in Color QuickDraw */
blend       = 32, /* replace destination pixel with a blend of the source
                     and destination pixel colors; if the destination is a
                     bitmap or 1-bit pixel map, revert to srcCopy mode */
addPin      = 33, /* replace destination pixel with the sum of the source
                     and destination pixel colors--up to a maximum
                     allowable value; if the destination is a bitmap or
                     1-bit pixel map, revert to srcBic mode */
addOver     = 34, /* replace destination pixel with the sum of the source
                     and destination pixel colors--but if the value of
                     the red, green, or blue component exceeds 65,536,
                     subtract 65,536 from that value; if the destination
                     is a bitmap or 1-bit pixel map, revert to srcXor
                     mode */
subPin      = 35, /* replace destination pixel with the difference of the
                     source and destination pixel colors--but not less
                     than a minimum allowable value; if the destination is
                     a bitmap or 1-bit pixel map, revert to srcOr mode */
addMax      = 37, /* compare the source and destination pixels, and
                     replace the destination pixel with the color
                     containing the greater saturation of each of the RGB
                     components; if the destination is a bitmap or 1-bit
                     pixel map, revert to srcBic mode */
subOver     = 38, /* replace destination pixel with the difference of the
                     source and destination pixel colors--but if the value
                     of the red, green, or blue component is less than 0,
                     add the negative result to 65,536; if the destination
                     is a bitmap or 1-bit pixel map, revert to
                     srcXor mode */
adMin       = 39, /* compare the source and destination pixels, and
                     replace the destination pixel with the color
                     containing the lesser saturation of each of the RGB
                     components; if the destination is a bitmap or 1-bit
                     pixel map, revert to srcOr mode */

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

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

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

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

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

};
```

Data Types

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

typedef unsigned char PixelType;
```

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

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

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

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

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

typedef struct MatchRec MatchRec;

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

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

struct CQDProcs {
   Ptr textProc;     /* text drawing */
   Ptr lineProc;     /* line drawing */
   Ptr rectProc;     /* rectangle drawing */
   Ptr rRectProc;    /* rounded rectangle drawing */
   Ptr ovalProc;     /* oval drawing */
   Ptr arcProc;      /* arc/wedge drawing */
   Ptr polyProc;     /* polygon drawing */
   Ptr rgnProc;      /* region drawing */
   Ptr bitsProc;     /* bit transfer */
   Ptr commentProc;  /* picture comment processing */
   Ptr txMeasProc;   /* text width measurement */
   Ptr getPicProc;   /* picture retrieval */
   Ptr putPicProc;   /* picture saving */
   Ptr opcodeProc;   /* reserved for future use */
   Ptr newProc1;     /* reserved for future use */
   Ptr newProc2;     /* reserved for future use */
   Ptr newProc3;     /* reserved for future use */
   Ptr newProc4;     /* reserved for future use */
   Ptr newProc5;     /* reserved for future use */
   Ptr newProc6;     /* reserved for future use */
};
typedef struct CQDProcs CQDProcs;
typedef CQDProcs *CQDProcsPtr;

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

Color QuickDraw Functions

Opening and Closing Color Graphics Ports

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

Managing a Color Graphics Pen

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

Changing the Background Pixel Pattern

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

Drawing With Color QuickDraw Colors

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

Determining Current Colors and Best Intermediate Colors

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

Calculating Color Fills

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

Creating, Setting, and Disposing of Pixel Maps

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

Creating and Disposing of Pixel Patterns

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

Creating and Disposing of Color Tables

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

Retrieving Color QuickDraw Result Codes

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

Customizing Color QuickDraw Operations

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

Reporting Data Structure Changes to QuickDraw

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

Application-Defined Function

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

Assembly-Language Summary

Data Structures

PixMap Data Structure

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

CGrafPort Data Structure

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

Relative Offsets of Additional Fields in a CGrafPort Record

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

RGBColor Data Structure

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

ColorSpec Data Structure

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

ColorTable Data Structure

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

MatchRec Data Structure

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

PixPat Data Structure

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

CQDProcs Data Structure

0	textProc	long	pointer to text-drawing routine
4	lineProc	long	pointer to line-drawing routine
8	rectProc	long	pointer to rectangle-drawing routine
12	rRectProc	long	pointer to rounded rectangle–drawing routine
16	ovalProc	long	pointer to oval-drawing routine
20	arcProc	long	pointer to arc/wedge-drawing routine
24	polyProc	long	pointer to polygon-drawing routine
28	rgnProc	long	pointer to region-drawing routine
32	bitsProc	long	pointer to bit transfer routine
36	commentProc	long	pointer to picture comment–processing routine
40	txMeasProc	long	pointer to text-width measurement routine
44	getPicProc	long	pointer to picture retrieval routine
48	putPicProc	long	pointer to picture-saving routine
52	opcodeProc	long	reserved for future use
56	newProc1	long	reserved for future use
60	newProc2	long	reserved for future use
64	newProc3	long	reserved for future use
68	newProc4	long	reserved for future use
72	newProc5	long	reserved for future use
76	newProc6	long	reserved for future use

GrafVars Data Structure

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

Trap Macros Requiring Routine Selectors

_QDExtensions

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

Result Codes

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

Graphics Devices

Contents

About Graphics Devices 5-3
Using Graphics Devices 5-6
  Optimizing Your Images for Different Graphics Devices 5-8
  Zooming Windows on Multiscreen Systems 5-9
  Setting a Device's Pixel Depth 5-13
  Exceptional Cases When Working With Color Devices 5-13
Graphics Devices Reference 5-14
  Data Structures 5-15
  Routines for Graphics Devices 5-19
    Creating, Setting, and Disposing of GDevice Records 5-19
    Getting the Available Graphics Devices 5-25
    Determining the Characteristics of a Video Device 5-29
    Changing the Pixel Depth for a Video Device 5-33
  Application-Defined Routine 5-35
  Resource 5-37
    The Screen Resource 5-37
Summary of Graphics Devices 5-38
  Pascal Summary 5-38
    Constants 5-38
    Data Types 5-39
    Routines for Graphics Devices 5-40
    Application-Defined Routine 5-40
  C Summary 5-41
    Constants 5-41
    Data Types 5-41
    Functions for Graphics Devices 5-43
    Application-Defined Function 5-44
  Assembly-Language Summary 5-44
    Data Structure 5-44
    Global Variables 5-44

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

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

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

About Graphics Devices

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

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

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

All existing `GDevice` records are linked together in a list, called the **device list;** the global variable `DeviceList` holds a handle to the first record in the list. At any given time, exactly one graphics device is the **current device** (also called the *active device*)—the one on which drawing is actually taking place. A handle to its `GDevice` record is stored in the global variable `TheGDevice`. By default, the `GDevice` record corresponding to the first video device found is marked as the current device; all other graphics devices in the list are initially marked as inactive.

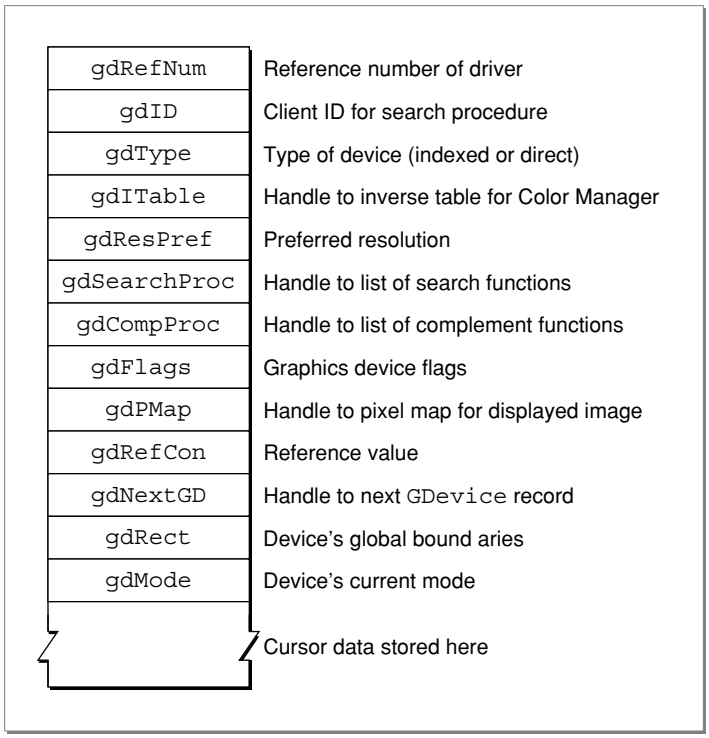
When the user moves a window or creates a window on another screen, and your application draws into that window, QuickDraw automatically makes the video device for that screen the current device. Color QuickDraw stores that information in the global variable `TheGDevice`. As Color QuickDraw draws across a user's video devices, it keeps switching to the `GDevice` record for the video device on which Color QuickDraw is actively drawing.

The user can use the Monitors control panel to set the desired pixel depth of each video device; to set the display to color, grayscale, or black and white; and to set the position of each screen relative to the **main screen** (that is, the one that contains the menu bar). The Monitors control panel stores all configuration information for a multiscreen system in the System file in a resource of type `'scrn'` that has a resource ID of 0. Your application should never create this resource, and should never alter or examine it. The `'scrn'` resource consists of an array of data structures that are analogous to `GDevice` records. Each element of this array contains information about a different video device.

When the `InitGraf` procedure (described in the chapter "Basic QuickDraw" in this book) initializes QuickDraw, it checks the System file for the `'scrn'` resource. If the `'scrn'` resource is found and it matches the hardware, `InitGraf` organizes the video devices according to the contents of this resource; if not, then QuickDraw uses only the video device for the startup screen.

The `GDevice` record is diagrammed in Figure 5-1. Some aspects of its contents are discussed after the figure; see page 5-15 for a complete description of the fields. Your application can use the routines described in this chapter to manipulate values for the fields in this record.

Figure 5-1 The `GDevice` record



The `gdITable` field points to an inverse table, which the Color Manager creates and maintains. An **inverse table** is a special Color Manager data structure arranged in such a manner that, given an arbitrary RGB color, its pixel value (that is, its index number in the CLUT) can be found quickly. The process is very fast once the table is built, but, if a color is changed in the video device's CLUT, the Color Manager must rebuild the inverse table the next time it has to find a color. The Color Manager is described in the chapter "Color Manager" in *Inside Macintosh: Advanced Color Imaging*.

The `gdPMap` field contains a handle to the pixel map that reflects the imaging capabilities of the graphics device. The pixel map's `PixelType` and `PixelSize` fields indicate whether the graphics device is direct or indexed and what pixel depth it displays. Color QuickDraw automatically synchronizes this pixel map's color table with the CLUT on the video device.

The `gdRect` field describes the graphics device's boundary rectangle in global coordinates. Color QuickDraw maps the (0,0) origin point of the global coordinate plane to the main screen's upper-left corner, and other screens are positioned adjacent to the main screen according to the settings made by the user with the Monitors control panel.

Using Graphics Devices

To use graphics devices, your application generally uses the QuickDraw routines described elsewhere in this book to draw images into a window; Color QuickDraw automatically displays your images in a manner appropriate for each graphics device that contains a portion of that window.

Note
The pixel map for a window's color graphics port always consists of the pixel depth, color table, and boundary rectangle of the main screen, even if the window is created on or moved to an entirely different screen. ◆

Instead of drawing directly into an onscreen graphics port, your application can use an offscreen graphics world (described in the chapter "Offscreen Graphics Worlds") to create images with the ideal pixel depth and color table required by your application. Then your application can use the `CopyBits` procedure to copy the images to the screen. Color QuickDraw converts the colors of the images for appropriate display on grayscale graphics devices and on direct and indirect color graphics devices. The manner in which Color QuickDraw translates the colors specified by your application to different graphics devices is described in the chapter "Color QuickDraw." However, if Color QuickDraw were to translate the colors of a color wheel (such as that used by the Color Picker, described in *Inside Macintosh: Advanced Color Imaging*), the image would appear as solid black on a black-and-white screen.

Many applications can let Color QuickDraw manage multiple video devices of differing dimensions and pixel depths. If your application needs more control over video device management—if it needs certain pixel depths or sets of colors to function effectively, for example—you can take several steps.

- If you need to know about the characteristics of available video devices, your application can use the `GetDeviceList` function to obtain a handle to the first `GDevice` record in the device list, the `GetGDevice` function to obtain a handle to the `GDevice` record for the current device, the `GetMainDevice` function to obtain a handle to the `GDevice` record for the main screen, or the `GetMaxDevice` function to obtain a handle to the `GDevice` record for the graphics device with the greatest pixel depth. Your application can then pass this handle to a routine like the `TestDeviceAttribute` function or the `HasDepth` function to determine various characteristics of a video device, or your application can examine the `gdRect` field of the `GDevice` record to determine the dimensions of the screen it represents.
- If you want to optimize your application's drawing for the best possible display on whatever type of screen is the current device, your application can use the `DeviceLoop` procedure, described on page 5-29, to determine the capabilities of the current device before drawing into a window on that device.
- If the current device is not suitable for the proper display of an image—for example, if the user has moved the window for your multicolored display of national flags to a black-and-white screen—your application can display the best image possible and display a message explaining that a more capable screen is required for better presentation of the image. Your application can use the `DeviceLoop` procedure to determine the capabilities of the current device.
- If your application uses the `HasDepth` function to determine that the current device can support the pixel depth required for the proper display of your image, but the `DeviceLoop` procedure indicates that the user has changed the screen's display, your application can use the `SetDepth` function to change the pixel depth of the screen. Note that the `SetDepth` function is provided for applications that are able to run only on graphics devices of a particular depth. Your application should use it only after soliciting the user's permission with a dialog box.
- If your application needs more control over colors on different indexed devices, your application can use the Palette Manager to arrange different sets of colors for particular images. Because the CLUT is variable on most video devices, your application can display up to 16 million colors, although on an 8-bit indexed device, for example, only 256 different colors can appear at once. See the chapter "Palette Manager" in *Inside Macintosh: Advanced Color Imaging* for more information.
- If your application needs to work with offscreen images that have characteristics different from those on the available graphics devices, your application can create offscreen graphics worlds, which contain their own `GDevice` records. See the chapter "Offscreen Graphics Worlds" in this book for more information.

To use the routines described in this chapter, your application must check for the existence of Color QuickDraw by using the `Gestalt` function with the `gestaltQuickDrawVersion` selector. The `Gestalt` function returns a 4-byte value in its `response` parameter; the low-order word contains QuickDraw version data. In that low-order word, the high-order byte gives the major revision number and the low-order byte gives the minor revision number. If the value returned in the `response` parameter is greater than or equal to the value of the constant `gestalt32BitQD`, then the system supports Color QuickDraw and all of the routines described in this chapter.

Optimizing Your Images for Different Graphics Devices

The `DeviceLoop` procedure searches for graphics devices that intersect your window's drawing region, and it informs your application of each different graphics device it finds. The `DeviceLoop` procedure provides your application with information about the current device's pixel depth and other attributes. Your application can then choose what drawing technique to use for the current device. For example, your application might use inversion to achieve a highlighting effect on a 1-bit graphics device, and, by using the `HiliteColor` procedure described in the chapter "Color QuickDraw," it might specify a color like magenta as the highlight color on a color graphics device.

For example, you can call `DeviceLoop` after calling the Event Manager procedure `BeginUpdate` whenever your application needs to draw into a window, as shown in Listing 5-1.

Listing 5-1 Using the `DeviceLoop` procedure

```
PROCEDURE DoUpdate (window: WindowPtr);
VAR
   windowType := Integer;
   myWindow: LongInt;
BEGIN
   windowType := MyGetWindowType(window);
   CASE windowType OF
   kSimpleRectanglesWindow: {simple case: window with 2 color rectangles}
      BEGIN
         BeginUpdate(window);
         myWindow := LongInt(window); {coerce window ptr for MyDrawingProc}
         DeviceLoop(window^.visRgn, @MyTrivialDrawingProc,
                    myWindow, []);
         EndUpdate;
      END;
   {handle other window types--documents, dialog boxes, etc.--here}
END;
```

When you use the `DeviceLoop` procedure, you must supply a handle to a drawing region and a pointer to your own application-defined drawing procedure. In Listing 5-1, a handle to the window's visible region and a pointer to an application-defined drawing procedure called `MyTrivialDrawingProc` are passed to `DeviceLoop`. For each graphics device it finds as the application updates its window, `DeviceLoop` calls `MyTrivialDrawingProc`.

Because `DeviceLoop` provides your drawing procedure with the pixel depth of the current device (along with other attributes passed to your drawing procedure in the `deviceFlags` parameter), your drawing procedure can optimize its drawing for whatever type of video device is the current device, as illustrated in Listing 5-2.

Listing 5-2 Drawing into different screens

```
PROCEDURE MyTrivialDrawingProc (depth: Integer;
                                deviceFlags: Integer;
                                targetDevice: GDHandle;
                                userData: LongInt);
VAR
   window: WindowPtr;
BEGIN
   window:= WindowPtr(userData);
   EraseRect(window^.portRect);
   CASE depth OF
   1:                              {black-and-white screen}
      MyDraw1BitRects(window);     {draw with ltGray, dkGray pats}
   2:
      MyDraw2BitRects(window); {draw with 2 of 4 available colors}
   {handle other screen depths here}
END;
```

Zooming Windows on Multiscreen Systems

The zoom box in the upper-right corner of the standard document window allows the user to alternate quickly between two window positions and sizes: the user state and the standard state.

The **user state** is the window size and location established by the user. If your application does not supply an initial user state, the user state is simply the size and location of the window when it was created, until the user resizes it.

The **standard state** is the window size and location that your application considers most convenient, considering the function of the document and the screen space available. In a word-processing application, for example, a standard-state window might show a full page, if possible, or a page of full width and as much length as fits on the screen. If the user changes the page size with the Page Setup command, the application might

adjust the standard state to reflect the new page size. If your application does not define a standard state, the Window Manager automatically sets the standard state to the entire gray region on the main screen, minus a three-pixel border on all sides. (See *Macintosh Human Interface Guidelines* for a detailed description of how your application determines where to open and zoom windows.) The user cannot change a window's standard state. (The user and standard states are stored in a data structure of type `WStateData` whose handle appears in the `dataHandle` field of the window record.)

Listing 5-3 illustrates an application-defined procedure, `DoZoomWindow`, which an application might call when the user clicks the zoom box. Because the user might have moved the window to a different screen since it was last zoomed, the procedure first determines which screen contains the largest area of the window and then calculates the ideal window size for that screen before zooming the window.

The screen calculations in the `DoZoomWindow` procedure compare `GDevice` records stored in the device list. (If Color QuickDraw is not available, `DoZoomWindow` assumes that it's running on a computer with a single screen.)

Listing 5-3 Zooming a window

```
PROCEDURE DoZoomWindow (thisWindow: windowPtr; zoomInOrOut: Integer);
VAR
   gdNthDevice, gdZoomOnThisDevice: GDHandle;
   savePort:                        GrafPtr;
   windRect, zoomRect, theSect:     Rect;
   sectArea, greatestArea:          LongInt;
   wTitleHeight:                    Integer;
   sectFlag:                        Boolean;
BEGIN
   GetPort(savePort);
   SetPort(thisWindow);
   EraseRect(thisWindow^.portRect);    {erase to avoid flicker}
   IF zoomInOrOut = inZoomOut THEN     {zooming to standard state}
   BEGIN
      IF NOT gColorQDAvailable THEN    {assume a single screen and }
      BEGIN                            { set standard state to full screen}
         zoomRect := screenBits.bounds;
         InsetRect(zoomRect, 4, 4);
         WStateDataHandle(WindowPeek(thisWindow)^.dataHandle)^^.stdState
                                                              := zoomRect;
      END
      ELSE                    {locate window on available screens}
      BEGIN
         windRect := thisWindow^.portRect;
         LocalToGlobal(windRect.topLeft);    {convert to global coordinates}
```

```
LocalToGlobal(windRect.botRight);
{calculate height of window's title bar}
wTitleHeight := windRect.top - 1 -
            WindowPeek(thisWindow)^.strucRgn^^.rgnBBox.top;
windRect.top := windRect.top - wTitleHeight;
gdNthDevice := GetDeviceList;    {get the first screen}
greatestArea := 0;         {initialize area to 0}
{check window against all gdRects in gDevice list and remember }
{ which gdRect contains largest area of window}
WHILE gdNthDevice <> NIL DO
IF TestDeviceAttribute(gdNthDevice, screenDevice) THEN
   IF TestDeviceAttribute(gdNthDevice, screenActive) THEN
   BEGIN
      {The SectRect function calculates the intersection }
      { of the window rectangle and this GDevice's boundary }
      { rectangle and returns TRUE if the rectangles intersect, }
      { FALSE if they don't.}
      sectFlag := SectRect(windRect, gdNthDevice^^.gdRect,
                           theSect);
      {determine which screen holds greatest window area}
      {first, calculate area of rectangle on current screen}
      WITH theSect DO
         sectArea := LongInt(right - left) * (bottom - top);
      IF sectArea > greatestArea THEN
      BEGIN
         greatestArea := sectArea;  {set greatest area so far}
         gdZoomOnThisDevice := gdNthDevice;  {set zoom device}
      END;
      gdNthDevice := GetNextDevice(gdNthDevice);   {get next }
   END;  {of WHILE}                                { GDevice record}
{if gdZoomOnThisDevice is on main device, allow for menu bar height}
IF gdZoomOnThisDevice = GetMainDevice THEN
   wTitleHeight := wTitleHeight + GetMBarHeight;
WITH gdZoomOnThisDevice^^.gdRect DO    {create the zoom rectangle}
BEGIN
   {set the zoom rectangle to the full screen, minus window title }
   { height (and menu bar height if necessary), inset by 3 pixels}
   SetRect(zoomRect, left + 3, top + wTitleHeight + 3,
          right - 3, bottom - 3);
   {If your application has a different "most useful" standard }
   { state, then size the zoom window accordingly.}
```

```
            {set up the WStateData record for this window}
            WStateDataHandle(WindowPeek(thisWindow)^.dataHandle)^^.stdState
                                                                  := zoomRect;
         END;
      END;
   END; {of inZoomOut}
   {if zoomInOrOut = inZoomIn, just let ZoomWindow zoom to user state}
   {zoom the window frame}
   ZoomWindow(thisWindow, zoomInOrOut, (thisWindow = FrontWindow));
   MyResizeWindow(thisWindow);   {application-defined window-sizing routine}
   SetPort(savePort);
END; (of DoZoomWindow)
```

If the user is zooming the window to the standard state, `DoZoomWindow` calculates a new standard size and location based on the application's own considerations, the current location of the window, and the available screens. The `DoZoomWindow` procedure always places the standard state on the screen where the window is currently displayed or, if the window spans screens, on the screen containing the largest area of the window.

Listing 5-3 uses the QuickDraw routines `GetDeviceList`, `TestDeviceAttribute`, `GetNextDevice`, `SectRect`, and `GetMainDevice` to examine characteristics of the available screens as stored in `GDevice` records. Most of the code in Listing 5-3 is devoted to determining which screen should display the window in the standard state.

IMPORTANT

Never use the `bounds` field of a `PixMap` record to determine the size of the screen; instead use the value of the `gdRect` field of the `GDevice` record for the screen, as shown in Listing 5-3. ▲

After calculating the standard state, if necessary, `DoZoomWindow` calls the `ZoomWindow` procedure to redraw the window frame in the new size and location and then calls the application-defined procedure `MyResizeWindow` to redraw the window's content region. For more information on zooming and resizing windows, see the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

Setting a Device's Pixel Depth

The Monitors control panel is the user interface for changing the pixel depth, color capabilities, and positions of video devices. Since the user can control the capabilities of the video device, your application should be flexible: although it may have a preferred pixel depth, your application should do its best to accommodate less than ideal conditions.

Your application can use the `SetDepth` function to change the pixel depth of a video device, but your application should do so only with the consent of the user. If your application must have a specific pixel depth, it can display a dialog box that offers the user a choice between changing to that depth or canceling display of the image. This dialog box saves the user the trouble of going to the Monitors control panel before returning to your application. (See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for more information about creating and using dialog boxes.)

Before calling `SetDepth`, use the `HasDepth` function to determine whether the available hardware can support the pixel depth you require. The `SetDepth` function is described on page 5-34, and the `HasDepth` function is described on page 5-33.

Exceptional Cases When Working With Color Devices

If your application always specifies colors in `RGBColor` records, Color QuickDraw automatically handles the colors on both indexed and direct devices. However, if your application does not specify colors in `RGBColor` records, your application may need to create and use special-purpose `CGrafPort`, `PixMap`, and `GDevice` records with the routines described in the chapter "Offscreen Graphics Worlds."

If your application must work with `CGrafPort`, `PixMap`, and `GDevice` records in ways beyond the scope of the routines described elsewhere in this book, the following guidelines may aid you in adapting Color QuickDraw to color graphics devices.

- Don't draw directly to the screen. Create your own offscreen graphics world (as described in the chapter "Offscreen Graphics Worlds") and use the `CopyBits`, `CopyMask`, or `CopyDeepMask` routine (described in the chapter "Color QuickDraw") to transfer the image to the screen.
- Don't directly change the `fgColor` or `bkColor` fields of a `GrafPort` record and expect them to be used as the pixel values. Color QuickDraw recalculates these values for each graphics device. If you want to draw with a color with a particular *index value,* use a palette with explicit colors, as described in *Inside Macintosh: Advanced Color Imaging*. For device-independent colors, use the `RGBForeColor` and `RGBBackColor` procedures, described in the chapter "Color QuickDraw" in this book.

- Don't copy a `GDevice` record's `PixMap` record. Instead, use the `NewPixMap` function or the `CopyPixMap` procedure, and fill all the fields. (These routines are described in the chapter "Color QuickDraw.") The `NewPixMap` function returns a `PixMap` record that is cloned from the `PixMap` record pointed to by the global variable `TheGDevice`. If you don't want a copy of the main screen's `PixMap` record—for example, you want one that is a different pixel depth—then you must fill out more fields than just `pixelSize`: you must fill out the `pixelType`, `cmpCount`, and `cmpSize` fields. Set the `pmVersion` field to 0 when initializing your own `PixMap` record. For future compatibility you should also set the `packType`, `packSize`, `planeBytes`, and `pmReserved` fields to 0. Don't assume a `PixMap` record has a color table—a pixel map for a direct device doesn't need one. For compatibility, a `PixMap` record for a direct device should have a dummy handle in the `pmTable` field that points to a `ColorTable` record with a seed value equal to `cmpSize` × `cmpCount` and a `ctSize` field set to 0.
- Fill out all the fields of a new `GDevice` record. When creating an offscreen `GDevice` record by calling `NewGDevice` with the `mode` parameter set to –1, you must fill out the fields of the `GDevice` record (for instance, the `gdType` field) yourself. If you want a copy of an existing `GDevice` record, copy the `gdType` field from it. If you explicitly want an indexed device, assign the `clutType` constant to the `gdType` field.

Graphics Devices Reference

This section describes the `GDevice` record, the routines that manipulate `GDevice` records, and the `'scrn'` resource.

"Data Structures" shows the Pascal data structure for the `GDevice` record, which contains information about a video device or offscreen graphics world. "Data Structures" also shows the data structure for the `DeviceLoopFlags` data type, which defines a set of options you can specify to the `DeviceLoop` procedure.

"Routines for Graphics Devices" describes routines for creating, setting, and disposing of `GDevice` records; getting the available graphics devices; and determining device characteristics. Your application generally never needs to create, set, or dispose of `GDevice` records. However, you may find it useful for your application to get `GDevice` records to determine the capabilities of the user's screens. When zooming a window, for example, your application can use `GDevice` records to determine which screen contains the largest area of a window, and then determine the ideal window size for that screen. You may also wish to use the `DeviceLoop` procedure, described in this chapter, if you want to optimize your application's drawing for graphics devices with different capabilities. "Application-Defined Routine" describes how you can define your own drawing procedure when optimizing your application's drawing for different graphics devices.

"Resource" describes the screen (`'scrn'`) resource. System software automatically creates and uses this resource; your application never needs it. The screen resource is documented here for your general information.

Data Structures

This section shows the Pascal data structure for the `GDevice` record, which can contain information about a video device or an offscreen graphics world. This section also shows the data structure for the `DeviceLoopFlags` data type, which defines a set of options you can specify to the `DeviceLoop` procedure.

GDevice

Color QuickDraw stores state information for video devices and offscreen graphics worlds in `GDevice` records. When the system starts up, it allocates and initializes one handle to a `GDevice` record for each video device it finds. When you use the `NewGWorld` function (described in the chapter "Offscreen Graphics Worlds" in this book), Color QuickDraw automatically creates a `GDevice` record for the new offscreen graphics world. The system links these `GDevice` records in a list, called the *device list*. (You can find a handle to the first element in the device list in the global variable `DeviceList`.) By default, the `GDevice` record corresponding to the first video device found is marked as the current device; all other graphics devices in the list are initially marked as inactive.

Note
Printing graphics ports, described in the chapter "Printing Manager" in this book, do not have `GDevice` records. ◆

When the user moves a window or creates a window on another screen, and your application draws into that window, Color QuickDraw automatically makes the video device for that screen the current device. Color QuickDraw stores that information in the global variable `TheGDevice`.

`GDevice` records that correspond to video devices have drivers associated with them. These drivers can be used to change the mode of the video device from black and white to color and to change the pixel depth. The set of routines supported by a video driver is defined and described in *Designing Cards and Drivers for the Macintosh Family*, third edition. Application-created `GDevice` records usually don't require drivers.

A `GDevice` record is defined as follows:

```
TYPE GDevice =
RECORD
   gdRefNum:      Integer;       {reference number of screen }
                                 { driver}
   gdID:          Integer;       {reserved; set to 0}
   gdType:        Integer;       {device type--indexed or direct}
   gdITable:      ITabHandle;    {handle to inverse table for }
                                 { Color Manager}
   gdResPref:     Integer;       {preferred resolution}
```

```
   gdSearchProc:  SProcHndl;     {handle to list of search }
                                 { functions}
   gdCompProc:    CProcHndl;     {handle to list of complement }
                                 { functions}
   gdFlags:       Integer;       {graphics device flags}
   gdPMap:        PixMapHandle;  {handle to PixMap record for }
                                 { displayed image}
   gdRefCon:      LongInt;       {reference value}
   gdNextGD:      GDHandle;      {handle to next graphics device}
   gdRect:        Rect;          {graphics device's global bounds}
   gdMode:        LongInt;       {graphics device's current mode}
   gdCCBytes:     Integer;       {width of expanded cursor data}
   gdCCDepth:     Integer;       {depth of expanded cursor data}
   gdCCXData:     Handle;        {handle to cursor's expanded }
                                 { data}
   gdCCXMask:     Handle;        {handle to cursor's expanded }
                                 { mask}
   gdReserved:    LongInt;       {reserved for future use--must }
                                 { be 0}
END;
```

Field descriptions

`gdRefNum` The reference number of the driver for the screen associated with the video device. For most video devices, this information is set at system startup time.

`gdID` Reserved. If you create your own `GDevice` record, set this field to 0.

`gdType` The general type of graphics device. Values include

```
CONST
clutType = 0;      {CLUT device--that is, one with }
                   { colors mapped with a color }
                   { lookup table}
fixedType = 1;     {fixed colors--that is, the }
                   { color lookup table can't }
                   { be changed}
directType = 2;    {direct RGB colors}
```

These types are described in more detail in the chapter "Color Manager" in *Inside Macintosh: Advanced Color Imaging*.

`gdITable` A handle to the inverse table for color mapping; the inverse table is described in the chapter "Color Manager" in *Inside Macintosh: Advanced Color Imaging*.

`gdResPref`	The preferred resolution for inverse tables.
`gdSearchProc`	A handle to the list of search functions, as described in the chapter "Color Manager" in *Inside Macintosh: Advanced Color Imaging*; its value is `NIL` for the default function.
`gdCompProc`	A handle to a list of complement functions, as described in the chapter "Color Manager" in *Inside Macintosh: Advanced Color Imaging*; its value is `NIL` for the default function.
`gdFlags`	The `GDevice` record's attributes. To set the attribute bits in the `gdFlags` field, use the `SetDeviceAttribute` procedure (described on page 5-22)—do not set these flags directly in the `GDevice` record. The constants representing each bit are listed here.

```
CONST {flag bits for gdFlags field of GDevice record}
gdDevType      = 0;  {if bit is set to 0, graphics device is }
                     { black and white; if set to 1, }
                     { graphics device supports color}
burstDevice    = 7;  {if bit is set to 1, graphics device }
                     { supports block transfer}
ext32Device    = 8;  {if bit is set to 1, graphics device }
                     { must be used in 32-bit mode}
ramInit        = 10; {if bit is set to 1, graphics device has }
                     { been initialized from RAM}
mainScreen     = 11; {if bit is set to 1, graphics device is }
                     { the main screen}
allInit        = 12; {if bit is set to 1, all graphics devices }
                     { were initialized from 'scrn' resource}
screenDevice   = 13; {if bit is set to 1, graphics device is }
                     { a screen}
noDriver       = 14; {if bit is set to 1, GDevice }
                     { record has no driver}
screenActive   = 15; {if bit is set to 1, graphics device is }
                     { active}
```

`gdPMap`	A handle to a `PixMap` record giving the dimension of the image buffer, along with the characteristics of the graphics device (resolution, storage format, color depth, and color table). `PixMap` records are described in the chapter "Color QuickDraw" in this book. For `GDevice` records, the high bit of the global variable `TheGDevice^^.gdPMap^^.pmTable^^.ctFlags` is always set.
`gdRefCon`	A value used by system software to pass device-related parameters. Since a graphics device is shared, you shouldn't store data here.
`gdNextGD`	A handle to the next graphics device in the device list. If this is the last graphics device in the device list, the field contains 0.

`gdRect`	The boundary rectangle of the graphics device represented by the `GDevice` record. The main screen has the upper-left corner of the rectangle set to (0,0). All other graphics devices are relative to this point.
`gdMode`	The current setting for the graphics device mode. This value is passed to the video driver to set its pixel depth and to specify color or black and white; applications don't need this information. See *Designing Cards and Drivers for the Macintosh Family,* third edition, for more information about the modes specified in this field.
`gdCCBytes`	The `rowBytes` value of the expanded cursor. Your application should not change this field. Cursors are described in the chapter "Cursor Utilities."
`gdCCDepth`	The depth of the expanded cursor. Your application should not change this field.
`gdCCXData`	A handle to the cursor's expanded data. Your application should not change this field.
`gdCCXMask`	A handle to the cursor's expanded mask. Your application should not change this field.
`gdReserved`	Reserved for future expansion; it must be set to 0 for future compatibility.

Your application should never need to directly change the fields of a `GDevice` record. If you find it absolutely necessary for your application to so, immediately use the `GDeviceChanged` procedure to notify Color QuickDraw that your application has changed the `GDevice` record. The `GDeviceChanged` procedure is described in the chapter "Color QuickDraw" in this book.

DeviceLoopFlags

When you use the `DeviceLoop` procedure (described on page 5-29), you can change its default behavior by using the `flags` parameter to specify one or more members of the set of flags defined by the `DeviceLoopFlags` data type. These flags are described here; if you want to use the default behavior of `DeviceLoop`, pass in the `flags` parameter 0 in your C code or an empty set ([ ]) in your Pascal code.

```
TYPE DeviceLoopFlags =
SET OF                {for flags parameter of DeviceLoop}
   (singleDevices,    {DeviceLoop doesn't group similar graphics }
                      { devices when calling drawing procedure}
   dontMatchSeeds,    {DeviceLoop doesn't consider ctSeed fields }
                      { of ColorTable records for graphics }
                      { devices when comparing them}
   allDevices);       {DeviceLoop ignores value of drawingRgn }
                      { parameter--instead, it calls drawing }
                      { procedure for every screen}
```

Field descriptions

`singleDevices` If this flag is not set, `DeviceLoop` calls your drawing procedure only once for each set of similar graphics devices, and the first one found is passed as the target device. (It is assumed to be representative of all the similar graphics devices.) If you set the `singleDevices` flag, then `DeviceLoop` does not group similar graphics devices—that is, those having identical pixel depths, black-and-white or color settings, and matching color table seeds—when it calls your drawing procedure.

`dontMatchSeeds`

If you set the `dontMatchSeeds` flag, then `DeviceLoop` doesn't consider color table seeds when comparing graphics devices for similarity; `DeviceLoop` ignores this flag if you set the `singleDevices` flag. Used primarily by the Palette Manager, the `ctSeed` field of a `ColorTable` record is described in the chapter "Color QuickDraw" in this book.

`allDevices` If you set the `allDevices` flag, `DeviceLoop` ignores the `drawingRgn` parameter and calls your drawing procedure for every graphics device. The value of current graphics port's `visRgn` field is not affected when you set this flag.

Routines for Graphics Devices

This section describes routines for creating, setting, and disposing of `GDevice` records; for getting the available video devices and offscreen graphics worlds; and for determining the characteristics of video devices and offscreen graphics worlds. Generally, your application won't need to use the routines for creating, setting, and disposing of `GDevice` records, because Color QuickDraw calls them automatically as appropriate. However, you may wish to use the other routines described in this section, particularly if you want to optimize your application's drawing for screens with different capabilities.

Creating, Setting, and Disposing of GDevice Records

Color QuickDraw uses `GDevice` records to maintain information about video devices and offscreen graphics worlds. A `GDevice` record must be allocated with the `NewGDevice` function and initialized with the `InitGDevice` procedure. Normally, your application does not call these routines directly. When the system starts up, it allocates and initializes one handle to a `GDevice` record for each video device it finds. When you use the `NewGWorld` function (described in the chapter "Offscreen Graphics Worlds" in this book), Color QuickDraw automatically creates a `GDevice` record for the new offscreen graphics world.

Whenever QuickDraw routines are used to draw into a graphics port on a video device, Color QuickDraw uses the `SetGDevice` procedure to make the video device for that screen the current device. Your application won't generally need to use this procedure, because when your application draws into a window on one or more screens, Color QuickDraw automatically switches `GDevice` records as appropriate; and when your application needs to draw into an offscreen graphics world, it can use the `SetGWorld` procedure to set the graphics port as well as the `GDevice` record for the offscreen environment. However, if your application uses the `SetPort` procedure (described in the chapter "Basic QuickDraw" in this book) instead of the `SetGWorld` procedure to set the graphics port to or from an offscreen graphics world, then your application must use `SetGDevice` in conjunction with `SetPort`.

You use the `SetDeviceAttribute` procedure to set attribute bits in a `GDevice` record.

When Color QuickDraw no longer needs a `GDevice` record, it uses the `DisposeGDevice` procedure to dispose of it. As with the other routines described in this section, your application typically does not need to use `DisposeGDevice`.

NewGDevice

You can use the `NewGDevice` function to create a new `GDevice` record, although you generally don't need to, because Color QuickDraw uses this function to create `GDevice` records for your application automatically.

```
FUNCTION NewGDevice (refNum: Integer; mode: LongInt): GDHandle;
```

`refNum`	Reference number of the graphics device for which you are creating a `GDevice` record. For most video devices, this information is set at system startup.
`mode`	The device configuration mode. Used by the screen driver, this value sets the pixel depth and specifies color or black and white.

DESCRIPTION

For the graphics device whose driver is specified in the `refNum` parameter and whose mode is specified in the `mode` parameter, the `NewGDevice` function allocates a new `GDevice` record and all of its handles, and then calls the `InitGDevice` procedure to initialize the record. As its function result, `NewGDevice` returns a handle to the new `GDevice` record. If the request is unsuccessful, `NewGDevice` returns `NIL`.

The `NewGDevice` function allocates the new `GDevice` record and all of its handles in the system heap, and the `NewGDevice` function sets all attributes in the `gdFlags` field of the `GDevice` record to `FALSE`. If your application creates a `GDevice` record, it can use the `SetDeviceAttribute` procedure, described on page 5-22, to change the flag bits in the `gdFlags` field of the `GDevice` record to `TRUE`. Your application should never directly change the `gdFlags` field of the `GDevice` record; instead, your application should use only the `SetDeviceAttribute` procedure.

If your application creates a `GDevice` record without a driver, it should set the `mode` parameter to –1. In this case, `InitGDevice` cannot initialize the `GDevice` record, so your application must perform all initialization of the record. A `GDevice` record's default mode is defined as 128; this is assumed to be a black-and-white mode. If you specify a value other than 128 in the `mode` parameter, the record's `gdDevType` bit in the `gdFlags` field of the `GDevice` record is set to `TRUE` to indicate that the graphics device is capable of displaying color.

The `NewGDevice` function doesn't automatically insert the `GDevice` record into the device list. In general, your application shouldn't create `GDevice` records, and if it ever does, it should never add them to the device list.

SPECIAL CONSIDERATIONS

If your program uses `NewGDevice` to create a graphics device without a driver, `InitGDevice` does nothing; instead, your application must initialize all fields of the `GDevice` record. After your application initializes the color table for the `GDevice` record, your application should call the Color Manager procedure `MakeITable` to build the inverse table for the graphics device.

The `NewGDevice` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

SEE ALSO

The `GDevice` record is described on page 5-15. See *Designing Cards and Drivers for the Macintosh Family,* third edition, for more information about the device modes that you can specify in the `mode` parameter. The Color Manager is described in *Inside Macintosh: Advanced Color Imaging*.

InitGDevice

The `NewGDevice` function uses the `InitGDevice` procedure to initialize a `GDevice` record.

```
PROCEDURE InitGDevice (gdRefNum: Integer; mode: LongInt;
                       gdh: GDHandle);
```

`gdRefNum`	Reference number of the graphics device. System software sets this number at system startup time for most graphics devices.
`mode`	The device configuration mode. Used by the screen driver, this value sets the pixel depth and specifies color or black and white.
`gdh`	The handle, returned by the `NewGDevice` function, to the `GDevice` record to be initialized.

DESCRIPTION

The `InitGDevice` procedure initializes the `GDevice` record specified in the `gdh` parameter. The `InitGDevice` procedure sets the graphics device whose driver has the reference number specified in the `gdRefNum` parameter to the mode specified in the `mode` parameter. The `InitGDevice` procedure then fills out the `GDevice` record, previously created with the `NewGDevice` function, to contain all information describing that mode.

The `mode` parameter determines the configuration of the device; possible modes for a device can be determined by interrogating the video device's ROM through Slot Manager routines. The information describing the device's mode is primarily contained in the video device's ROM. If the video device has a fixed color table, then that table is read directly from the ROM. If the video device has a variable color table, then `InitGDevice` uses the default color table defined in a `'clut'` resource, contained in the System file, that has a resource ID equal to the video device's pixel depth.

In general, your application should never need to call `InitGDevice`. All video devices are initialized at start time, and users change modes through the Monitors control panel.

SPECIAL CONSIDERATIONS

If your program uses `NewGDevice` to create a graphics device without a driver, `InitGDevice` does nothing; instead, your application must initialize all fields of the `GDevice` record. After your application initializes the color table for the `GDevice` record, your application should call the Color Manager procedure `MakeITable` to build the inverse table for the graphics device.

The `InitGDevice` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

The `GDevice` record is described on page 5-15. See *Designing Cards and Drivers for the Macintosh Family,* third edition, for more information about the device modes that you can specify in the `mode` parameter. The `MakeITable` procedure is described in the chapter "Color Manager" in *Inside Macintosh: Advanced Color Imaging*.

SetDeviceAttribute

To set the attribute bits of a `GDevice` record, use the `SetDeviceAttribute` procedure.

```
PROCEDURE SetDeviceAttribute (gdh: GDHandle; attribute: Integer;
                              value: Boolean);
```

`gdh` A handle to a `GDevice` record.

`attribute` One of the following constants, which represent bits in the `gdFlags` field of a `GDevice` record:

```
CONST {flag bits for gdFlags field of GDevice record}
 gdDevType      = 0;  {if bit is set to 0, graphics }
                      { device is black and white; }
                      { if set to 1, device supports }
                      { color}
 burstDevice    = 7;  {if bit is set to 1, device }
                      { supports block transfer}
 ext32Device    = 8;  {if bit is set to 1, device }
                      { must be used in 32-bit mode}
 ramInit        = 10; {if bit is set to 1, device has }
                      { been initialized from RAM}
 mainScreen     = 11; {if bit is set to 1, device is }
                      { the main screen}
 allInit        = 12; {if bit is set to 1, all }
                      { devices were initialized from }
                      { 'scrn' resource}
 screenDevice   = 13; {if bit is set to 1, device is }
                      { a screen}
 noDriver       = 14; {if bit is set to 1, GDevice }
                      { record has no driver}
 screenActive   = 15; {if bit is set to 1, device is }
                      { active}
```

`value` A value of either 0 or 1 for the flag bit specified in the `attribute` parameter.

DESCRIPTION

For the graphics device specified in the `gdh` parameter, the `SetDeviceAttribute` procedure sets the flag bit specified in the `attribute` parameter to the value specified in the `value` parameter.

SPECIAL CONSIDERATIONS

Your application should never directly change the `gdFlags` field of the `GDevice` record; instead, your application should use only the `SetDeviceAttribute` procedure.

The `SetDeviceAttribute` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SetGDevice

Your application can use the `SetGDevice` procedure to set a `GDevice` record as the current device.

```
PROCEDURE SetGDevice (gdh: GDHandle);
```

`gdh` A handle to a `GDevice` record.

DESCRIPTION

The `SetGDevice` procedure sets the specified `GDevice` record as the current device. Your application won't generally need to use this procedure, because when your application draws into a window on one or more screens, Color QuickDraw automatically switches `GDevice` records as appropriate; and when your application needs to draw into an offscreen graphics world, it can use the `SetGWorld` procedure to set the graphics port as well as the `GDevice` record for the offscreen environment. However, if your application uses the `SetPort` procedure (described in the chapter "Basic QuickDraw" in this book) instead of the `SetGWorld` procedure to set the graphics port to or from an offscreen graphics world, then your application must use `SetGDevice` in conjunction with `SetPort`.

A handle to the currently active device is kept in the global variable `TheGDevice`.

SPECIAL CONSIDERATIONS

The `SetGDevice` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

See the chapter "Offscreen Graphics Worlds" in this book for information about the `SetGWorld` procedure and about drawing into offscreen graphics worlds.

DisposeGDevice

Although your application generally should never need to use this routine, the `DisposeGDevice` procedure disposes of a `GDevice` record, releases the space allocated for it, and disposes of all the data structures allocated for it. The `DisposeGDevice` procedure is also available as the `DisposGDevice` procedure.

```
PROCEDURE DisposeGDevice (gdh: GDHandle);
```

`gdh` A handle to the `GDevice` record.

DESCRIPTION

The `DisposeGDevice` procedure disposes of a `GDevice` record, releases the space allocated for it, and disposes of all the data structures allocated for it. Color QuickDraw calls this procedure when appropriate.

SEE ALSO

When your application uses the `DisposeGWorld` procedure to dispose of an offscreen graphics world, `DisposeGDevice` disposes of its `GDevice` record. See the chapter "Offscreen Graphics Worlds" in this book for a description of `DisposeGWorld`.

Getting the Available Graphics Devices

To gain access to the `GDevice` record for a video device—for example, to determine the size and pixel depth of its attached screen—your application needs to get a handle to that record.

Your application can use the `GetDeviceList` function to obtain a handle to the first `GDevice` record in the device list, the `GetGDevice` function to obtain a handle to the `GDevice` record for the current device, the `GetMainDevice` function to obtain a handle to the `GDevice` record for the main screen, and the `GetMaxDevice` function to obtain a handle to the `GDevice` record for the video device with the greatest pixel depth.

All existing `GDevice` records are linked together in the device list. After using one of these functions to obtain a handle to one of the `GDevice` records in the device list, your application can use the `GetNextDevice` function to obtain a handle to the next `GDevice` record in the list.

Two related functions, `GetGWorld` and `GetGWorldDevice`, also allow you to obtain handles to `GDevice` records. To get the `GDevice` record for the current device, you can use the `GetGWorld` function. To get a handle to the `GDevice` record for a particular offscreen graphics world, you can use the `GetGWorldDevice` function. These two functions are described in the next chapter, "Offscreen Graphics Worlds."

GetGDevice

To obtain a handle to the `GDevice` record for the current device, use the `GetGDevice` function.

```
FUNCTION GetGDevice: GDHandle;
```

DESCRIPTION

The `GetGDevice` function returns a handle to the `GDevice` record for the current device. (At any given time, exactly one video device is the current device—that is, the one on which drawing is actually taking place.)

Color QuickDraw stores a handle to the current device in the global variable `TheGDevice`.

SPECIAL CONSIDERATIONS

The `GetGDevice` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

GetDeviceList

To obtain a handle to the first `GDevice` record in the device list, use the `GetDeviceList` function.

```
FUNCTION GetDeviceList: GDHandle;
```

DESCRIPTION

The `GetDeviceList` function returns a handle to the first `GDevice` record in the global variable `DeviceList`.

SPECIAL CONSIDERATIONS

The `GetDeviceList` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

SEE ALSO

Listing 5-3 on page 5-10 illustrates the use of this function.

GetMainDevice

To obtain a handle to the `GDevice` record for the main screen, use the `GetMainDevice` function.

```
FUNCTION GetMainDevice: GDHandle;
```

DESCRIPTION

The `GetMainDevice` function returns a handle to the `GDevice` record that corresponds to the main screen—that is, the one containing the menu bar.

A handle to the main device is kept in the global variable `MainDevice`.

SPECIAL CONSIDERATIONS

The `GetMainDevice` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

SEE ALSO

Listing 5-3 on page 5-10 illustrates the use of this function.

GetMaxDevice

To obtain a handle to the `GDevice` record for the video device with the greatest pixel depth, use the `GetMaxDevice` function.

```
FUNCTION GetMaxDevice (globalRect: Rect): GDHandle;
```

`globalRect`
: A rectangle, in global coordinates, that intersects the graphics devices that you are searching to find the one with the greatest pixel depth.

DESCRIPTION

As its function result, `GetMaxDevice` returns a handle to the `GDevice` record for the video device that has the greatest pixel depth among all graphics devices that intersect the rectangle you specify in the `globalRect` parameter.

SPECIAL CONSIDERATIONS

The `GetMaxDevice` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

GetNextDevice

After using the `GetDeviceList` function to obtain a handle to the first `GDevice` record in the device list, `GetGDevice` to obtain a handle to the `GDevice` record for the current device, `GetMainDevice` to obtain a handle to the `GDevice` record for the main screen, or `GetMaxDevice` to obtain a handle to the `GDevice` record for the video device with the greatest pixel depth, you can use the `GetNextDevice` function to obtain a handle to the next `GDevice` record in the list.

```
FUNCTION GetNextDevice (curDevice: GDHandle): GDHandle;
```

`curDevice` A handle to the `GDevice` record at which you want the search to begin.

DESCRIPTION

The `GetNextDevice` function returns a handle to the next `GDevice` record in the device list. If there are no more `GDevice` records in the list, it returns `NIL`.

SPECIAL CONSIDERATIONS

The `GetNextDevice` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

SEE ALSO

Listing 5-3 on page 5-10 illustrates the use of this function.

Determining the Characteristics of a Video Device

For drawing images that are optimized for every screen they cross, your application can use the `DeviceLoop` procedure. The `DeviceLoop` procedure searches for graphics devices that intersect your window's drawing region, and it calls your drawing procedure for each different video device it finds. The `DeviceLoop` procedure provides your drawing procedure with information about the current device's pixel depth and other attributes.

To determine whether the flag bit for an attribute has been set in the `gdFlags` field of a `GDevice` record, your application can use the `TestDeviceAttribute` function.

To determine whether a video device supports a specific pixel depth, your application can also use the `HasDepth` function, described on page 5-33. To change the pixel depth of a video device, your application can use the `SetDepth` function, described on page 5-34.

If you need to determine the resolution of the main device, you can use the `ScreenRes` procedure.

DeviceLoop

For drawing images that are optimized for every screen they cross, use the `DeviceLoop` procedure.

```
PROCEDURE DeviceLoop (drawingRgn: RgnHandle;
                      drawingProc: DeviceLoopDrawingProcPtr;
                      userData: LongInt; flags: DeviceLoopFlags);
```

`drawingRgn`
A handle to the region in which you will draw; this drawing region uses coordinates that are local to its graphics port.

`drawingProc`
A pointer to your own drawing procedure.

`userData` Any additional data that you wish to supply to your drawing procedure.

`flags` One or more members of the set of flags defined by the `DeviceLoopFlags` data type:

```
TYPE
   DeviceLoopFlags = SET OF
   (singleDevices,dontMatchSeeds,allDevices);
```

These flags are described in the following text; if you want to use the default behavior of `DeviceLoop`, specify an empty set ([]) in this parameter.

DESCRIPTION

The `DeviceLoop` procedure searches for graphics devices that intersect your window's drawing region, and it calls your drawing procedure for each video device it finds. In the `drawingRgn` parameter, supply a handle to the region in which you wish to draw; in the `drawingProc` parameter, supply a pointer to your drawing procedure. In the `flags` parameter, you can specify members of the set of these flags defined by the `DeviceLoopFlags` data type:

`singleDevices`	If this flag is not set, `DeviceLoop` calls your drawing procedure only once for each set of similar graphics devices, and the first one found is passed as the target device. (It is assumed to be representative of all the similar graphics devices.) If you set the `singleDevices` flag, then `DeviceLoop` does not group similar graphics devices—that is, those having identical pixel depths, black-and-white or color settings, and matching color table seeds—when it calls your drawing procedure.
`dontMatchSeeds`	If you set the `dontMatchSeeds` flag, then `DeviceLoop` doesn't consider the `ctSeed` field of `ColorTable` records for graphics devices when comparing them; `DeviceLoop` ignores this flag if you set the `singleDevices` flag.
`allDevices`	If you set the `allDevices` flag, `DeviceLoop` ignores the `drawingRgn` parameter and calls your drawing procedure for every device. The value of current graphics port's `visRgn` field is not affected when you set this flag.

For each dissimilar video device that intersects this region, `DeviceLoop` calls your drawing procedure. For example, after a call to the Event Manager procedure `BeginUpdate`, the region you specify in the `drawingRgn` parameter can be the same as the visible region for the active window. Because `DeviceLoop` provides your drawing procedure with the pixel depth and other attributes of each video device, your drawing procedure can optimize its drawing for each video device—for example, by using the `HiliteColor` procedure to set magenta as the highlight color on a color video device.

SPECIAL CONSIDERATIONS

The `DeviceLoop` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

SEE ALSO

Listing 5-1 on page 5-8 illustrates the use of `DeviceLoop`. See page 5-35 for a description of the drawing procedure you must provide for the `drawingProc` parameter. Offscreen graphics worlds are described in the next chapter. The `HiliteColor` procedure is described in the chapter "Color QuickDraw" in this book.

TestDeviceAttribute

To determine whether the flag bit for an attribute has been set in the `gdFlags` field of a `GDevice` record, use the `TestDeviceAttribute` function.

```
FUNCTION TestDeviceAttribute (gdh: GDHandle;
                              attribute: Integer): Boolean;
```

`gdh` A handle to a `GDevice` record.

`attribute` One of the following constants, which represent bits in the `gdFlags` field of a `GDevice` record:

```
CONST {flag bits for gdFlags field of GDevice record}
 gdDevType      = 0;  {if bit is set to 0, graphics }
                      { device is black and white; }
                      { if set to 1, device supports }
                      { color}
 burstDevice    = 7;  {if bit is set to 1, device }
                      { supports block transfer}
 ext32Device    = 8;  {if bit is set to 1, device }
                      { must be used in 32-bit mode}
 ramInit        = 10; {if bit is set to 1, device has }
                      { been initialized from RAM}
 mainScreen     = 11; {if bit is set to 1, device is }
                      { the main screen}
 allInit        = 12; {if bit is set to 1, all }
                      { devices were initialized from }
                      { 'scrn' resource}
 screenDevice   = 13; {if bit is set to 1, device is }
                      { a screen}
 noDriver       = 14; {if bit is set to 1, GDevice }
                      { record has no driver}
 screenActive   = 15; {if bit is set to 1, device is }
                      { active}
```

DESCRIPTION

The `TestDeviceAttribute` function tests a single graphics device attribute to see if its bit is set to 1 and, if so, returns `TRUE`. Otherwise, `TestDeviceAttribute` returns `FALSE`.

SPECIAL CONSIDERATIONS

The `TestDeviceAttribute` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

SEE ALSO

Listing 5-3 on page 5-10 illustrates the use of `TestDeviceAttribute`. Your application can use the `SetDeviceAttribute` procedure, described on page 5-22, to change any of the flags tested by the `TestDeviceAttribute` function.

ScreenRes

If you need to determine the resolution of the main device, you can use the `ScreenRes` procedure.

```
PROCEDURE ScreenRes (VAR scrnHRes,scrnVRes: Integer);
```

DESCRIPTION

In the `scrnHRes` parameter, the `ScreenRes` procedure returns the number of horizontal pixels per inch displayed by the current device. In the `scrnVRes` parameter, it returns the number of vertical pixels per inch.

To determine the resolutions of all available graphics devices, you should examine their `GDevice` records (described on page 5-15). The horizontal and vertical resolutions for a graphics device are stored in the `hRes` and `vRes` fields, respectively, of the `PixMap` record for the device's `GDevice` record.

SPECIAL CONSIDERATIONS

Currently, QuickDraw and the Printing Manager always assume a screen resolution of 72 dpi.

Do not use the actual screen resolution as a scaling factor when drawing into a printing graphics port; instead, always use 72 dpi as the scaling factor. See the chapter "Printing Manager" in this book for more information about the Printing Manager and drawing into a printing graphics port.

ASSEMBLY-LANGUAGE INFORMATION

The horizontal resolution, in pixels per inch, is stored in the global variable `ScrHRes`, and the vertical resolution is stored in the global variable `ScrVRes`.

Changing the Pixel Depth for a Video Device

The Monitors control panel is the user interface for changing the pixel depth, color capabilities, and positions of video devices. Since the user can control the capabilities of the video device, your application should be flexible: although it may have a preferred pixel depth, your application should do its best to accommodate less than ideal conditions.

If it is absolutely necessary for your application to draw on a video device of a specific pixel depth, your application can use the `SetDepth` function to change its pixel depth. Before calling `SetDepth`, use the `HasDepth` function to determine whether the available hardware can support the pixel depth you require.

HasDepth

To determine whether a video device supports a specific pixel depth, you can use the `HasDepth` function.

```
FUNCTION HasDepth (aDevice: GDHandle; depth: Integer;
                   whichFlags: Integer; flags: Integer): Integer;
```

`aDevice` A handle to the `GDevice` record of the video device.

`depth` The pixel depth for which you're testing.

`whichFlags`
The `gdDevType` constant, which represents a bit in the `gdFlags` field of the `GDevice` record. (If this bit is set to 0 in the `GDevice` record, the video device is black and white; if the bit is set to 1, the device supports color.)

`flags` The value 0 or 1. If you pass 0 in this parameter, the `HasDepth` function tests whether the video device is black and white; if you pass 1 in this parameter, `HasDepth` tests whether the video device supports color.

DESCRIPTION

The `HasDepth` function checks whether the video device you specify in the `aDevice` parameter supports the pixel depth you specify in the `depth` parameter, and whether the device is black and white or color, whichever you specify in the `flags` parameter.

The `HasDepth` function returns 0 if the device does not support the depth you specify in the `depth` parameter or the display mode you specify in the `flags` parameter.

Any other value indicates that the device supports the specified depth and display mode. The function result contains the mode ID that QuickDraw passes to the video driver to set its pixel depth and to specify color or black and white. You can pass this mode ID in the `depth` parameter for the `SetDepth` function (described next) to set the graphics device to the pixel depth and display mode for which you tested.

SPECIAL CONSIDERATIONS

The `HasDepth` function may move or purge blocks of memory in the application heap. Your application should not call this function at interrupt time.

SEE ALSO

See *Designing Cards and Drivers for the Macintosh Family,* third edition, for more information about the device modes returned as a function result for `HasDepth`.

SetDepth

To change the pixel depth of a video device, use the `SetDepth` function.

```
FUNCTION SetDepth (aDevice: GDHandle; depth: Integer;
                   whichFlags: Integer; flags: Integer): OSErr;
```

`aDevice` A handle to the `GDevice` record of the video device whose pixel depth you wish to change.

`depth` The mode ID returned by the `HasDepth` function (described in the previous section) indicating that the video device supports the desired pixel depth. Alternatively, you can pass the desired pixel depth directly in this parameter, although you should use the `HasDepth` function to ensure that the device supports this depth.

`whichFlags`
The `gdDevType` constant, which represents a bit in the `gdFlags` field of the `GDevice` record. (If this bit is set to 0 in the `GDevice` record, the video device is black and white; if the bit is set to 1, the device supports color.)

`flags` The value 0 or 1. If you pass 0 in this parameter, the `SetDepth` function changes the video device to black and white; if you pass 1 in this parameter, `SetDepth` changes the video device to color.

DESCRIPTION

The `SetDepth` function sets the video device you specify in the `aDevice` parameter to the pixel depth you specify in the `depth` parameter, and it sets the device to either black and white or color as you specify in the `flags` parameter. You should use the `HasDepth` function to ensure that the video device supports the values you specify to `SetDepth`. The `SetDepth` returns zero if successful, or it returns a nonzero value if it cannot impose the desired depth and display mode on the requested device.

The `SetDepth` function does not change the `'scrn'` resource; when the system is restarted, the original depth for this device is restored.

SPECIAL CONSIDERATIONS

Your application should use `SetDepth` only if your application can run on devices of a particular pixel depth and is unable to adapt to any other depth. Your application should display a dialog box that offers the user a choice between changing to that depth or canceling display of the image before your application uses `SetDepth`. Such a dialog box saves the user the trouble of going to the Monitors control panel before returning to your application.

The `SetDepth` function may move or purge blocks of memory in the application heap. Your application should not call this function at interrupt time.

SEE ALSO

See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information about creating and using dialog boxes.

Application-Defined Routine

Your application can use the `DeviceLoop` procedure (described on page 5-29) before drawing images that are optimized for every screen they cross. The `DeviceLoop` procedure searches for video devices that intersect your drawing region, and it calls a drawing procedure that you define for every different video device it finds.

For each video device that intersects a drawing region that you define (generally, the update region of a window), `DeviceLoop` calls your drawing procedure. Because `DeviceLoop` provides your drawing procedure with the pixel depth and other attributes of the current device, your drawing procedure can optimize its drawing for whatever type of graphics device is the current device. When highlighting, for example, your application might invert black and white when drawing onto a 1-bit video device but use magenta as the highlight color when drawing onto a color video device. In this case, even were your window to span both a black-and-white and a color screen, the user sees the selection inverted on the black-and-white screen, while magenta would be used to highlight the selection on the color screen.

You must provide a pointer to your drawing procedure in the `drawingProc` parameter for `DeviceLoop`.

MyDrawingProc

Here's how to declare a drawing procedure to supply to the `DeviceLoop` procedure if you were to name the procedure `MyDrawingProc`:

```
PROCEDURE MyDrawingProc (depth: Integer; deviceFlags: Integer;
                         targetDevice: GDHandle;
                         userData: LongInt);
```

`depth` The pixel depth of the graphics device.

`deviceFlags`
Any of the following constants, which represent bits that are set to 1 in the `gdFlags` field of the `GDevice` record (described on page 5-15) for the current device:

```
CONST {flag bits for gdFlags field of GDevice record}
 gdDevType      = 0;  {if bit is set to 1, graphics }
                      { device supports color}
 burstDevice    = 7;  {if bit is set to 1, device }
                      { supports block transfer}
 ext32Device    = 8;  {if bit is set to 1, device }
                      { must be used in 32-bit mode}
 ramInit        = 10; {if bit is set to 1, device has }
                      { been initialized from RAM}
 mainScreen     = 11; {if bit is set to 1, device is }
                      { the main screen}
 allInit        = 12; {if bit is set to 1, all }
                      { devices were initialized from }
                      { 'scrn' resource}
 screenDevice   = 13; {if bit is set to 1, device is }
                      { a screen}
 noDriver       = 14; {if bit is set to 1, GDevice }
                      { record has no driver}
 screenActive   = 15; {if bit is set to 1, device is }
                      { active}
```

`targetDevice`
A handle to the `GDevice` record (described on page 5-15) for the current device.

`userData` A value that your application supplies to the `DeviceLoop` procedure, which in turn passes the value to your drawing procedure for whatever purpose you deem useful.

DESCRIPTION

Your drawing procedure should analyze the pixel depth passed in the `depth` parameter and the values passed in the `deviceFlags` parameter, and then draw in a manner that is optimized for the current device.

SEE ALSO

Listing 5-2 on page 5-9 illustrates a simple drawing procedure called by `DeviceLoop`.

Resource

The user can use the Monitors control panel to set the desired pixel depth of each screen; whether it displays color, grayscale, or black and white; and the position of each screen relative to the main screen. The Monitors control panel stores all configuration information for a multiscreen system in the System file in a resource of type `'scrn'` that has a resource ID of 0. Your application should never create this resource, and should never alter or examine it.

When the `InitGraf` procedure (described in the chapter "Basic QuickDraw" in this book) initializes Color QuickDraw, it checks the System file for the `'scrn'` resource. If the `'scrn'` resource is found and it matches the hardware, `InitGraf` organizes the video devices according to the contents of this resource; if not, then Color QuickDraw uses only the video device for the startup screen.

The Screen Resource

The `'scrn'` resource consists of an array of data structures that are analogous to `GDevice` records. Each data structure in this array contains information about a different video device. Because your application shouldn't create or alter the `'scrn'` resource, its structure is not described here.

Summary of Graphics Devices

Pascal Summary

Constants

```
CONST
   {flag bits for gdType field of GDevice record}
   clutType = 0;     {CLUT device--that is, one with colors mapped with a }
                     { color lookup table}
   fixedType = 1;    {fixed colors--that is, the color lookup table }
                     { can't be changed}
   directType = 2;   {direct RGB colors}

   {flag bits for gdFlags field of GDevice record}
   gdDevType      = 0;  {if bit is set to 0, graphics device is black }
                        { and white; if bit is set to 1, graphics device }
                        { supports color}
   burstDevice    = 7;  {if bit is set to 1, graphics device supports block }
                        { transfer}
   ext32Device    = 8;  {if bit is set to 1, graphics device must be used }
                        { in 32-bit mode}
   ramInit        = 10; {if bit is set to 1, graphics device has been }
                        { initialized from RAM}
   mainScreen     = 11; {if bit is set to 1, graphics device is the main }
                        { screen}
   allInit        = 12; {if bit is set to 1, all graphics devices were }
                        { initialized from 'scrn' resource}
   screenDevice   = 13; {if bit is set to 1, graphics device is a screen}
   noDriver       = 14; {if bit is set to 1, GDevice record has no driver}
   screenActive   = 15; {if bit is set to 1, graphics device is current }
                        { device}
```

Data Types

```
TYPE
GDHandle    = ^GDPtr;
GDPtr       = ^GDevice;
GDevice     =
RECORD
   gdRefNum:      Integer;       {reference number of screen driver}
   gdID:          Integer;       {reserved; set to 0}
   gdType:        Integer;       {type of device--indexed or direct}
   gdITable:      ITabHandle;    {handle to inverse table for Color Manager}
   gdResPref:     Integer;       {preferred resolution}
   gdSearchProc:  SProcHndl;     {handle to list of search functions}
   gdCompProc:    CProcHndl;     {handle to list of complement functions}
   gdFlags:       Integer;       {graphics device flags}
   gdPMap:        PixMapHandle;  {handle to PixMap record for displayed }
                                 { image}
   gdRefCon:      LongInt;       {reference value}
   gdNextGD:      GDHandle;      {handle to next graphics device}
   gdRect:        Rect;          {graphics device's boundary in global }
                                 { coordinates}
   gdMode:        LongInt;       {graphics device's current mode}
   gdCCBytes:     Integer;       {width of expanded cursor data}
   gdCCDepth:     Integer;       {depth of expanded cursor data}
   gdCCXData:     Handle;        {handle to cursor's expanded data}
   gdCCXMask:     Handle;        {handle to cursor's expanded mask}
   gdReserved:    LongInt;       {reserved for future use; must be 0}
END;

QDErr = Integer;

DeviceLoopDrawingProcPtr = ProcPtr;

DeviceLoopFlags = SET OF   {for flags parameter of DeviceLoop}
   (singleDevices,         {DeviceLoop doesn't group similar graphics }
                           { devices when calling drawing procedure}
   dontMatchSeeds,         {DeviceLoop doesn't consider ctSeed fields }
                           { of ColorTable records for graphics devices }
                           { when comparing them}
   allDevices);            {DeviceLoop ignores value of drawingRgn }
                           { parameter--instead, it calls drawing procedure }
                           { for every screen}
```

Routines for Graphics Devices

Creating, Setting, and Disposing of GDevice Records

```
{ DisposeGDevice is also spelled as DisposGDevice }
FUNCTION NewGDevice          (refNum: Integer; mode: LongInt): GDHandle;
PROCEDURE InitGDevice        (gdRefNum: Integer; mode: LongInt;
                              gdh: GDHandle);
PROCEDURE SetDeviceAttribute
                             (gdh: GDHandle; attribute: Integer;
                              value: Boolean);
PROCEDURE SetGDevice         (gdh: GDHandle);
PROCEDURE DisposeGDevice     (gdh: GDHandle);
```

Getting the Available Graphics Devices

```
FUNCTION GetGDevice          : GDHandle;
FUNCTION GetDeviceList       : GDHandle;
FUNCTION GetMainDevice       : GDHandle;
FUNCTION GetMaxDevice        (globalRect: Rect): GDHandle;
FUNCTION GetNextDevice       (curDevice: GDHandle): GDHandle;
```

Determining the Characteristics of a Video Device

```
PROCEDURE DeviceLoop         (drawingRgn: RgnHandle;
                              drawingProc: DeviceLoopDrawingProcPtr;
                              userData: LongInt; flags: DeviceLoopFlags);
FUNCTION TestDeviceAttribute
                             (gdh: GDHandle;
                              attribute: Integer): Boolean;
PROCEDURE ScreenRes          (VAR scrnHRes,scrnVRes: Integer);
```

Changing the Pixel Depth for a Video Device

```
FUNCTION HasDepth            (aDevice: GDHandle; depth: Integer;
                              whichFlags: Integer; flags: Integer): Integer;
FUNCTION SetDepth            (aDevice: GDHandle; depth: Integer;
                              whichFlags: Integer; flags: Integer): OSErr;
```

Application-Defined Routine

```
PROCEDURE MyDrawingProc      (depth: Integer; deviceFlags: Integer;
                              targetDevice: GDHandle; userData: LongInt);
```

C Summary

Constants

```
enum {
   /* flag bits for gdType field of GDevice record */
   clutType       = 0;  /* CLUT device--that is, one with colors mapped with
                           a color lookup table */
   fixedType      = 1;  /* fixed colors--that is, the color lookup table
                           can't be changed */
   directType     = 2;  /* direct RGB colors */

   /* flag bits for gdFlags field of GDevice record */
   gdDevType      = 0,  /* if bit is set to 0, graphics device is black and
                           white; if set to 1, device is color */
   burstDevice    = 7,  /* if bit is set to 1, graphics device supports block
                           transfer */
   ext32Device    = 8,  /* if bit is set to 1, graphics device must be used
                           in 32-bit mode */
   ramInit        = 10, /* if bit is set to 1, graphics device was
                           initialized from RAM */
   mainScreen     = 11, /* if bit is set to 1, graphics device is the main
                           screen */
   allInit        = 12, /* if bit is set to 1, all graphics devices were
                           initialized from 'scrn' resource */
   screenDevice   = 13, /* if bit is set to 1, graphics device is a screen
                           device */
   noDriver       = 14, /* if bit is set to 1, GDevice record has
                           no driver */
   screenActive   = 15, /* if bit is set to 1, graphics device is current
                           device */
};
```

Data Types

```
struct GDevice {
   short          gdRefNum;      /* reference number of screen driver */
   short          gdID;          /* reserved; set to 0 */
   short          gdType;        /* type of device--indexed or direct */
   ITabHandle     gdITable;      /* handle to inverse table for Color
                                    Manager */
   short          gdResPref;     /* preferred resolution */
```

```
   SProcHndl      gdSearchProc;  /* handle to list of search functions */
   CProcHndl      gdCompProc;    /* handle to list of complement functions */
   short          gdFlags;       /* graphics device flags */
   PixMapHandle   gdPMap;        /* handle to PixMap record for displayed
                                    image */
   long           gdRefCon;      /* reference value */
   GDHandle       gdNextGD;      /* handle to next graphics device */
   Rect           gdRect;        /* graphics device's boundary in global
                                    coordinates */
   long           gdMode;        /* graphics device's current mode */
   short          gdCCBytes;     /* width of expanded cursor data */
   short          gdCCDepth;     /* depth of expanded cursor data */
   Handle         gdCCXData;     /* handle to cursor's expanded data */
   Handle         gdCCXMask;     /* handle to cursor's expanded mask */
   long           gdReserved;    /* reserved for future use; must be 0 */
};
typedef struct GDevice GDevice;
typedef GDevice *GDPtr, **GDHandle;

typedef short QDErr;

typedef pascal void  (*DeviceLoopDrawingProcPtr)
                     (short depth, short deviceFlags,
                      GDHandle targetDevice, long userData);

/* for flags parameter of DeviceLoop */
enum  {singleDevicesBit = 0,dontMatchSeedsBit = 1,allDevicesBit = 2};
enum  {singleDevices = 1 << singleDevicesBit,   /* DeviceLoop doesn't group
                                                   similar graphics devices
                                                   when calling drawing
                                                   procedure */
      dontMatchSeeds = 1 << dontMatchSeedsBit,  /* DeviceLoop doesn't
                                                   consider ctSeed fields of
                                                   ColorTable records for
                                                   graphics devices when
                                                   comparing them */
      allDevices = 1 << allDevicesBit};         /* DeviceLoop ignores value
                                                   of drawingRgn parameter--
                                                   instead it calls drawing
                                                   procedure for every
                                                   screen */

typedef unsigned long DeviceLoopFlags;
```

Functions for Graphics Devices

Creating, Setting, and Disposing of GDevice Records

```
/* DisposeGDevice is also spelled as DisposGDevice */

pascal GDHandle NewGDevice  (short refNum, long mode);
pascal void InitGDevice     (short gdRefNum, long mode, GDHandle gdh);
pascal void SetDeviceAttribute
                            (GDHandle gdh, short attribute, Boolean value);
pascal void SetGDevice      (GDHandle gdh);
pascal void DisposeGDevice  (GDHandle gdh);
```

Getting the Available Graphics Devices

```
pascal GDHandle GetGDevice  (void);
pascal GDHandle GetDeviceList
                            (void);
pascal GDHandle GetMainDevice
                            (void);
pascal GDHandle GetMaxDevice
                            (const Rect *globalRect);
pascal GDHandle GetNextDevice
                            (GDHandle curDevice);
```

Determining the Characteristics of a Video Device

```
pascal void DeviceLoop      (RgnHandle drawingRgn,
                             DeviceLoopDrawingProcPtr drawingProc,
                             long userData, DeviceLoopFlags flags);
pascal Boolean TestDeviceAttribute
                            (GDHandle gdh, short attribute);
pascal void ScreenRes       (short *scrnHRes, short *scrnVRes);
```

Changing the Pixel Depth for a Video Device

```
pascal Integer HasDepth     (GDHandle aDevice, Integer depth,
                             Integer whichFlags, Integer flags);
pascal OSErr SetDepth       (GDHandle aDevice, Integer depth,
                             Integer whichFlags, Integer flags);
```

Application-Defined Function

```
pascal void MyDrawingProc   (Integer depth, Integer deviceFlags,
                             GDHandle targetDevice, LongInt userData);
```

Assembly-Language Summary

Data Structure

GDevice Data Structure

0	`gdRefNum`	word	`refNum` of screen driver
2	`gdID`	word	reserved; set to 0
4	`gdType`	word	general type of graphics device
6	`gdITable`	long	handle to inverse table
10	`gdResPref`	word	preferred resolution for inverse tables
12	`gdSearchProc`	long	search function pointer
16	`gdCompProc`	long	complement function pointer
20	`gdFlags`	word	graphics device flags word
22	`gdPMap`	long	handle to pixel map describing graphics device
26	`gdRefCon`	long	reference value
30	`gdNextGD`	long	handle to next `GDevice` record
34	`gdRect`	8 bytes	graphics device's bounds in global coordinates
42	`gdMode`	long	device's current mode
46	`gdCCBytes`	word	width of expanded cursor data
48	`gdCCDepth`	word	depth of expanded cursor data
50	`gdCCXData`	long	handle to cursor's expanded data
54	`gdCCXMask`	long	handle to cursor's expanded mask
58	`gdReserved`	long	reserved; must be 0

Global Variables

`DeviceList`	Handle to the first `GDevice` record in the device list.
`MainDevice`	Handle to the `GDevice` record for the main screen.
`ScrHRes`	The horizontal resolution, in pixels per inch, for the current device.
`ScrVRes`	The vertical resolution, in pixels per inch, for the current device.
`TheGDevice`	Handle to the `GDevice` record for the current device.

Offscreen Graphics Worlds

Contents

About Offscreen Graphics Worlds 6-3
Using Offscreen Graphics Worlds 6-4
  Creating an Offscreen Graphics World 6-5
  Setting the Graphics Port for an Offscreen Graphics World 6-8
  Drawing Into an Offscreen Graphics World 6-8
  Copying an Offscreen Image Into a Window 6-9
  Updating an Offscreen Graphics World 6-9
  Creating a Mask and a Source Image in Offscreen Graphics Worlds 6-10
Offscreen Graphics Worlds Reference 6-12
  Data Structures 6-12
  Routines 6-16
    Creating, Altering, and Disposing of Offscreen Graphics Worlds 6-16
    Saving and Restoring Graphics Ports and Offscreen Graphics Worlds 6-27
    Managing an Offscreen Graphics World's Pixel Image 6-30
Summary of Offscreen Graphics Worlds 6-40
  Pascal Summary 6-40
    Constants 6-40
    Data Types 6-41
    Routines 6-42
  C Summary 6-43
    Constants 6-43
    Data Types 6-44
    Functions 6-45
  Assembly-Language Summary 6-46
  Result Codes 6-46

This chapter describes QuickDraw routines and data structures that your application can use to create offscreen graphics worlds. Whether your application uses Color QuickDraw or basic QuickDraw, you should read this chapter to improve your application's appearance and performance when it draws onscreen images.

Read this chapter to learn how to set up and use an **offscreen graphics world**—a sophisticated environment for preparing complex color or black-and-white images before displaying them on the screen. Offscreen graphics worlds are available on all Macintosh computers that support System 7.

You can use all of the drawing operations described in the chapters "QuickDraw Drawing" and "Color QuickDraw" in this book to create images in an offscreen graphics world. After preparing an image in an offscreen graphics world, you can use the `CopyBits`, `CopyMask`, or `CopyDeepMask` procedure to move the image to an onscreen color graphics port or basic graphics port. Color graphics ports are described in the chapter "Color QuickDraw," and basic graphics ports are described in the chapter "Basic QuickDraw" in this book.

To support your application in preparing an offscreen image for display on a screen, Color QuickDraw by default uses the screen's `GDevice` record to define the pixel depth and color table for the offscreen graphics world. The `GDevice` record is described in the chapter "Graphics Devices" in this book.

Your application can treat an offscreen graphics world as a virtual screen where your application has complete control over its drawing environment, and on which your application can draw a complex image where the user can't see the various steps your application must take before completing it. For example, your application can use QuickDraw drawing routines to build a complex color image in an offscreen graphics world; then, after building the image, your application can use `CopyBits` to copy it quickly to the screen. This prevents the choppiness that could occur if your application were to construct the image directly in a color graphics port on the screen.

About Offscreen Graphics Worlds

An offscreen graphics world is defined by a private data structure that, in Color QuickDraw, contains a `CGrafPort` record and its handles to associated `PixMap` and `ColorTable` records. The offscreen graphics world also contains a reference to a `GDevice` record and other state information. On computers lacking Color QuickDraw, `GWorldPtr` points to an extension of the `GrafPort` record. An offscreen graphics world for a basic QuickDraw system does not contain a reference to a `GDevice` record, but it does support a special type of 1-bit pixel map. When your application uses the `NewGWorld` function to create an offscreen world, `NewGWorld` returns a pointer of type `GWorldPtr`, which your application uses to refer to the offscreen graphics world. This pointer is defined as follows:

```
TYPE GWorldPtr = CGrafPtr;
```

Offscreen graphics worlds have two primary purposes.

- They prevent any other application or desk accessory from changing your drawing environment while your application is creating an image. An offscreen graphics world that you create cannot be modified by any other application.
- They increase onscreen drawing speed and visual smoothness. For example, suppose your application draws multiple graphics objects in a window, and then needs to update part of that window. If your image is very complex, your application can copy it from an offscreen graphics world onto the screen faster than it can repeat all of the steps necessary to redraw the image onscreen. At the same time, your application avoids the choppy visual effect that arises from drawing a large number of separate objects.

The term *offscreen graphics world* implies that you prepare an image on the global coordinate plane somewhere outside the boundary rectangles for the user's screens. While this is possible, you more typically use the coordinates of the port rectangle for an onscreen window when preparing your "offscreen" image; until you copy the image to the onscreen window, the image in the offscreen graphics world is drawn into a part of memory not used by the video device and therefore remains hidden from the user.

Using Offscreen Graphics Worlds

To use an offscreen graphics world, you generally

- use the `NewGWorld` function to create an offscreen graphics world
- use the `GetGWorld` procedure to save the onscreen graphics port for the active window
- use the `SetGWorld` procedure to make the offscreen graphics world the current graphics port
- use the `LockPixels` function to prevent the base address for the offscreen pixel image from moving while you draw into it or copy from it
- use the `EraseRect` procedure to initialize the offscreen pixel image
- use the basic QuickDraw and Color QuickDraw routines described elsewhere in this book to draw into the offscreen graphics world
- use the `SetGWorld` procedure to restore the active window as the current graphics port
- use the `CopyBits` procedure to copy the image from the offscreen graphics world into the active window
- use the `UnlockPixels` procedure to allow the Memory Manager to move the base address for the offscreen pixel image
- use the `DisposeGWorld` procedure to dispose of all the memory allocated for an offscreen graphics world when you no longer need its offscreen pixel image

If you want to use the `CopyMask` or `CopyDeepMask` procedure, you can create another offscreen graphics world and draw your mask into that offscreen world.

These tasks are explained in greater detail in the rest of this chapter.

Before using the routines described in this chapter, you must use the `InitGraf` procedure, described in the chapter "Basic QuickDraw," to initialize QuickDraw. You should also ensure the availability of these routines by checking for the existence of System 7 or Color QuickDraw.

You can make sure that offscreen graphics world routines are available on any computer—including one supporting only basic QuickDraw—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 offscreen graphics worlds are supported.

You can also test for offscreen graphics world support by using the `Gestalt` function with the `gestaltQuickDrawVersion` selector. If the value returned in the `response` parameter is equal to or greater than the value of the constant `gestalt32BitQD`, then the system supports both Color QuickDraw and offscreen graphics worlds.

You can use the `Gestalt` function with the `gestaltQuickDrawVersion` selector to determine whether the user's system supports offscreen color pixel maps. If the bit indicated by the `gestaltHasDeepGWorlds` constant is set in the `response` parameter, then offscreen color pixel maps are available.

For more information about the `Gestalt` function, see the chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities*.

Creating an Offscreen Graphics World

You create an offscreen graphics world with the `NewGWorld` function. It creates a new offscreen graphics port, a new offscreen pixel map, and (on computers that support Color QuickDraw) either a new offscreen `GDevice` record or a link to an existing one. It returns a data structure of type `GWorldPtr` by which your application refers to your new offscreen graphics world. Listing 6-1 illustrates how to create an offscreen graphics world.

Listing 6-1 Using a single offscreen graphics world and the `CopyBits` procedure

```
PROCEDURE MyPaintRectsThruGWorld (wp: WindowPtr);
   VAR
      origPort:               GrafPtr;
      origDev:                GDHandle;
      myErr:                  QDErr;
      myOffGWorld:            GWorldPtr;
      offPixMapHandle:        PixMapHandle;
      good:                   Boolean;
      sourceRect, destRect:   Rect;
```

```
BEGIN
   GetGWorld(origPort, origDev);          {save window's graphics port}
   myErr := NewGWorld(myOffGWorld, 0,     {create offscreen graphics world, }
                    wp^.portRect,         { using window's port rectangle}
                    NIL, NIL, []);
   IF (myOffGWorld = NIL) OR (myErr <> noErr) THEN
      ; {handle error here}
   SetGWorld(myOffGWorld, NIL);  {make offscreen world the current port}
   offPixMapHandle := GetGWorldPixMap(myOffGWorld);   {get handle to }
   good := LockPixels(offPixMapHandle); { offscreen pixel image and lock it}
   IF NOT good THEN
      ; {handle error here}
   EraseRect(myOffGWorld^.portRect);         {initialize its pixel image}
   MyPaintAndFillColorRects; {paint a blue rectangle, fill a green rectangle}
   SetGWorld(origPort, origDev);          {make window the current port}
      {next, for CopyBits, create source and destination rectangles that }
      { exclude scroll bar areas}
   sourceRect := myOffGWorld^.portRect;   {use offscreen portRect for source}
   sourceRect.bottom := myOffGWorld^.portRect.bottom - 15;
   sourceRect.right := myOffGWorld^.portRect.right - 15;
   destRect := wp^.portRect;        {use window portRect for destination}
   destRect.bottom := wp^.portRect.bottom - 15;
   destRect.right := wp^.portRect.right - 15;
      {next, use CopyBits to transfer the offscreen image to the window}
   CopyBits(GrafPtr(myOffGWorld)^.portBits,  {coerce graphics world's }
                                             { PixMap to a BitMap}
            GrafPtr(wp)^.portBits,     {coerce window's PixMap to a BitMap}
            sourceRect, destRect, srcCopy, NIL);
   IF QDError <> noErr THEN
      ; {likely error is that there is insufficient memory}
   UnlockPixels(offPixMapHandle);         {unlock the pixel image}
   DisposeGWorld(myOffGWorld);            {dispose of offscreen world}
END;
```

When you use `NewGWorld`, you can specify a pixel depth, a boundary rectangle (which also becomes the port rectangle), a color table, a `GDevice` record, and option flags for memory allocation for the offscreen graphics world. Typically, however, you pass 0 as the pixel depth, a window's port rectangle as the offscreen world's boundary rectangle, `NIL` for both the color table and `GDevice` record, and an empty set ([ ]) in your Pascal code or 0 in your C code for the option flags. This provides your application with the default behavior of `NewGWorld`, and it supports computers running only basic QuickDraw. This also allows QuickDraw to optimize the `CopyBits`, `CopyMask`, and `CopyDeepMask` procedures when your application copies the image you create in an offscreen graphics world into the window's port rectangle.

When creating an offscreen graphics world, if you specify 0 as the pixel depth, the port rectangle for a window as the boundary rectangle, and no option flags, the `NewGWorld` function

- uses the pixel depth of the screen with the greatest pixel depth from among all screens intersected by the window
- aligns the pixel image to the screen for optimum performance for the `CopyBits` procedure
- uses the color table and `GDevice` record for the screen with the greatest pixel depth from among all screens intersected by the window
- allocates an unpurgeable base address for the offscreen pixel image in your application heap
- allows graphics accelerators to cache the offscreen pixel image

The application-defined routine `MyPaintRectsThruGWorld` in Listing 6-1, for example, specifies the default behavior for `NewGWorld`. The `MyPaintRectsThruGWorld` routine dereferences the window pointer passed in the `wp` parameter to obtain a window's port rectangle, which `MyPaintRectsThruGWorld` passes to `NewGWorld` as the boundary and port rectangle for the offscreen graphics world.

Setting the Graphics Port for an Offscreen Graphics World

Before drawing into the offscreen graphics port created in Listing 6-1 on page 6-5, `MyPaintRectsThruGWorld` saves the graphics port for the front window by calling the `GetGWorld` procedure, which saves the current graphics port and its `GDevice` record. Then `MyPaintRectsThruGWorld` makes the offscreen graphics world the current port by calling the `SetGWorld` procedure. After drawing into the offscreen graphics world, `MyPaintRectsThruGWorld` also uses `SetGWorld` to restore the active window as the current graphics port.

Instead of using the `GetPort` and `SetPort` procedures for saving and restoring offscreen graphics worlds, you must use `GetGWorld` and `SetGWorld`; you can also use `GetGWorld` and `SetGWorld` for saving and restoring color and basic graphics ports.

Drawing Into an Offscreen Graphics World

You must call the `LockPixels` function before drawing to or copying from an offscreen graphics world. The `LockPixels` function prevents the base address for an offscreen pixel image from being moved while you draw into it or copy from it.

If the base address for an offscreen pixel image hasn't been purged by the Memory Manager or if its base address is not purgeable, `LockPixels` returns `TRUE` as its function result, and your application can draw into or copy from the offscreen pixel image. However, if the base address for an offscreen pixel image has been purged, `LockPixels` returns `FALSE` to indicate that you cannot draw into it or copy from it. (At that point, your application should either call the `UpdateGWorld` function to reallocate the offscreen pixel image and then reconstruct it, or draw directly into an onscreen graphics port.)

After setting the offscreen graphics world to the current graphics port, `MyPaintRectsThruGWorld` in Listing 6-1 on page 6-5 uses the `GetGWorldPixMap` function to get a handle to an offscreen pixel map. Passing this handle to the `LockPixels` function, `MyPaintRectsThruGWorld` locks the memory for the offscreen pixel image in preparation for drawing into its pixel map.

IMPORTANT

On a system running only basic QuickDraw, the `GetGWorldPixMap` function returns the handle to a 1-bit pixel map that your application can supply as a parameter to `LockPixels` and the other routines related to offscreen graphics worlds that are described in this chapter. On a basic QuickDraw system, however, your application should not supply this handle to Color QuickDraw routines. ▲

The `MyPaintRectsThruGWorld` routine initializes the offscreen pixel image to all white by calling the `EraseRect` procedure, which is described in the chapter "Basic QuickDraw." The `MyPaintRectsThruGWorld` routine then calls another application-defined routine, `MyPaintAndFillColorRects`, to draw color rectangles into the pixel map for the offscreen graphics world.

IMPORTANT
You cannot dereference the `GWorldPtr` data structure to get to the pixel map. The `baseAddr` field of the `PixMap` record for an offscreen graphics world contains a handle instead of a pointer, which is what the `baseAddr` field for an onscreen pixel map contains. You must use the `GetPixBaseAddr` function (described on page 6-38) to obtain a pointer to the `PixMap` record for an offscreen graphics world. ▲

Copying an Offscreen Image Into a Window

After preparing an image in the offscreen graphics world, your application must use `SetGWorld` to restore the active window as the current graphics port, as illustrated in Listing 6-1 on page 6-5.

To copy the image from an offscreen graphics world into a window, use the `CopyBits` procedure. Specify the offscreen graphics world as the source image for `CopyBits`, and specify the window as its destination. When using `CopyBits`, you must coerce the offscreen graphics world's `GWorldPtr` data type to a data structure of type `GrafPtr`. Similarly, whenever a color graphics port is your destination, you must coerce the window's `CGrafPtr` data type to a data structure of type `GrafPtr`. (The `CopyBits` procedure is described in the chapter "QuickDraw Drawing.")

As long as you're drawing into an offscreen graphics world or copying the image out of it, you must leave its pixel image locked. When you are finished drawing into and copying from an offscreen graphics world, use the `UnlockPixels` procedure. To help prevent heap fragmentation, the `UnlockPixels` procedure allows the Memory Manager to move the base address for the offscreen pixel image. (For more information about Macintosh memory management, see *Inside Macintosh: Memory*.)

Finally, call the `DisposeGWorld` procedure when your application no longer needs the pixel image associated with this offscreen graphics world, as illustrated in Listing 6-1.

Updating an Offscreen Graphics World

When the user resizes or moves a window, changes the pixel depth of a screen that a window intersects, or modifies a color table, you can use the `UpdateGWorld` function to reflect the user's choices in the offscreen graphics world. The `UpdateGWorld` function, described on page 6-23, allows you to change the pixel depth, boundary rectangle, or color table for an existing offscreen graphics world without recreating it and redrawing its contents. You should also call `UpdateGWorld` after every update event.

Calling `UpdateGWorld` and then `CopyBits` when the user makes these changes helps your application get the maximum refresh speed when updating the window. See the chapters "Event Manager" and "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for more information about handling update events in windows and about resizing windows.

Creating a Mask and a Source Image in Offscreen Graphics Worlds

When you use the `CopyMask` or `CopyDeepMask` procedure (described in the chapter "QuickDraw Drawing"), you can create the source image and its mask in separate offscreen graphics worlds. Plates 3 and 4 at the front of this book illustrate how to use `CopyMask` in this way. The source image in Plate 3 consists of graduated gray stripes; this image is created in an offscreen graphics world. The mask in Plate 3 consists of a black rectangle alongside a red rectangle; this mask is created in a separate graphics world that shares the same coordinates as the source image.

When the `CopyMask` procedure copies the grayscale image through this mask, the result is the untitled window illustrated in the bottom half of Plate 3. The black pixels in the mask cause `CopyMask` to copy directly into the window those pixels from the source image that are masked by the black rectangle. The red pixels in the mask cause `CopyMask` to alter most of the source pixels masked by the red rectangle when copying them. That is, the source pixels that are completely black are changed to the mask's red when copied into the window. The source pixels that are completely white are left unaltered when copied into the window. The source pixels that are between black and white are given a graduated amount of the mask's red.

Listing 6-2 shows the code that produces the window shown in Plate 3.

Listing 6-2 Using two offscreen graphics worlds and the `CopyMask` procedure

```
PROCEDURE MyCopyBlackAndRedMasks (wp: WindowPtr);
   VAR
      origPort:                           GrafPtr;
      origDevice:                         GDHandle;
      myErr:                              QDErr;
      myOffScreen1, myOffScreen2:         GWorldPtr;
      theColor:                           RGBColor;
      i:                                  Integer;
      offPixMapHandle1, offPixMapHandle2: PixMapHandle;
      good:                               Boolean;
      myRect:                             Rect;
BEGIN
   GetGWorld(origPort, origDevice);       {save window's graphics port}
                        {create an offscreen world for building an image}
   myErr := NewGWorld(myOffScreen1, 0, wp^.portRect, NIL, NIL, []);
   IF (myOffScreen1 = NIL) OR (myErr <> noErr) THEN
      ; {handle error here}
                        {create another offscreen world for building a mask}
   myErr := NewGWorld(myOffScreen2, 0, wp^.portRect, NIL, NIL, []);
   IF (myOffScreen2 = NIL) OR (myErr <> noErr) THEN
      ; {handle error here}
```

```
   SetGWorld(myOffScreen1, NIL); {make first offscreen world the }
                                 { current port}
   offPixMapHandle1 := GetGWorldPixMap(myOffScreen1);
   good := LockPixels(offPixMapHandle1);  {lock its pixel image}
   IF NOT good THEN
      ; {handle error here}
   EraseRect(myOffScreen1^.portRect);     {initialize its pixel image}
   FOR i := 0 TO 9 DO      {draw graduated grayscale stripes for the image}
      BEGIN
         theColor.red := i * 7168;
         theColor.green := i * 7168;
         theColor.blue := i * 7168;
         RGBForeColor(theColor);
         SetRect(myRect, myOffScreen1^.portRect.left, i * 10,
                 myOffScreen1^.portRect.right, i * 10 + 10);
         PaintRect(myRect);
      END;
   SetGWorld(myOffScreen2, NIL); {make second offscreen world the }
                                 { current port}
   offPixMapHandle2 := GetGWorldPixMap(myOffScreen2);
   good := LockPixels(offPixMapHandle2);  {lock its pixel image}
   IF NOT good THEN
      ; {handle error here}
   EraseRect(myOffScreen2^.portRect);     {initialize its pixel image}
   SetRect(myRect, 20, 20, 80, 80);
   PaintRect(myRect);   {paint a black rectangle in the mask}
   SetRect(myRect, 100, 20, 160, 80);
   theColor.red := $FFFF;  theColor.green := $0000; theColor.blue := $0000;
   RGBForeColor(theColor);
   PaintRect(myRect);   {paint a red rectangle in the mask}
   SetGWorld(wp, GetMainDevice); {make window the current port}
   EraseRect(wp^.portRect);      {erase the window before using CopyMask}
   CopyMask(GrafPtr(myOffScreen1)^.portBits, {use gray image as source}
            GrafPtr(myOffScreen2)^.portBits, {use 2-rectangle image as mask}
            GrafPtr(wp)^.portBits,           {use window as destination}
            myOffScreen1^.portRect,
            myOffScreen2^.portRect,
            wp^.portRect);
   UnlockPixels(offPixMapHandle1);  UnlockPixels(offPixMapHandle2);
   DisposeGWorld(myOffScreen1);  DisposeGWorld(myOffScreen2);
   SetGWorld(origPort, origDevice);    {restore original graphics port}
END;
```

Offscreen Graphics Worlds Reference

This section describes the data structures and routines that your application can use to create and manage offscreen graphics worlds. "Data Structures" shows the Pascal data structures for the `GWorldPtr` and `GWorldFlags` data types. "Routines" describes routines for creating, altering, and disposing of offscreen graphics worlds; for saving and restoring offscreen graphics worlds; and for managing an offscreen graphics world's pixel map.

Data Structures

This section shows the Pascal data structures for the `GWorldPtr` and `GWorldFlags` data types. Your application uses pointers of type `GWorldPtr` to refer to the offscreen graphics worlds it creates. Several routines in this chapter expect or return values defined by the `GWorldFlags` data type.

GWorldPtr

An offscreen graphics world in Color QuickDraw contains a `CGrafPort` record—and its handles to associated `PixMap` and `ColorTable` records—that describes an offscreen graphics port and contains references to a `GDevice` record and other state information. The actual data structure for an offscreen graphics world is kept private to allow for future extensions. However, when your application uses the `NewGWorld` function to create an offscreen world, `NewGWorld` returns a pointer of type `GWorldPtr` by which your application refers to the offscreen graphics world. This pointer is defined as follows:

```
TYPE GWorldPtr = CGrafPtr;
```

On computers lacking Color QuickDraw, `GWorldPtr` points to an extension of the `GrafPort` record.

GWorldFlags

Several routines in this chapter expect or return values defined by the `GWorldFlags` data type, which is defined as follows:

```
TYPE GWorldFlags =
SET OF (
   pixPurge,          {specify to NewGWorld to make base address }
                      { for offscreen pixel image purgeable}
   noNewDevice,       {specify to NewGWorld to not create a new }
                      { GDevice record for offscreen world}
   useTempMem,        {specify to NewGWorld to create base }
                      { address for offscreen pixel image in }
                      { temporary memory}
   keepLocal,         {specify to NewGWorld to keep offscreen }
                      { pixel image in main memory}
   gWorldFlag4,       {reserved}
   gWorldFlag5,       {reserved}
   pixelsPurgeable,   {returned by GetPixelsState to indicate }
                      { that base address for offscreen pixel }
                      { image is purgeable; specify to }
                      { SetPixelsState to make base address for }
                      { pixel image purgeable}
   pixelsLocked,      {returned by GetPixelsState to indicate }
                      { that base address for offscreen pixel }
                      { image is locked; specify to }
                      { SetPixelsState to lock base address for }
                      { offscreen pixel image}
   gWorldFlag8,       {reserved}
   gWorldFlag9,       {reserved}
   gWorldFlag10,      {reserved}
   gWorldFlag11,      {reserved}
   gWorldFlag12,      {reserved}
   gWorldFlag13,      {reserved}
   gWorldFlag14,      {reserved}
   gWorldFlag15,      {reserved}
   mapPix,            {returned by UpdateGWorld if it remapped }
                      { colors to a new color table}
   newDepth,          {returned by UpdateGWorld if it translated }
                      { pixel map to a different pixel depth}
   alignPix,          {returned by UpdateGWorld if it realigned }
                      { pixel image to onscreen window}
   newRowBytes,       {returned by UpdateGWorld if it changed }
                      { rowBytes field of PixMap record}
```

```
   reallocPix,       {returned by UpdateGWorld if it reallocated }
                     { base address for offscreen pixel image}
   gWorldFlag21,     {reserved}
   gWorldFlag22,     {reserved}
   gWorldFlag23,     {reserved}
   gWorldFlag24,     {reserved}
   gWorldFlag25,     {reserved}
   gWorldFlag26,     {reserved}
   gWorldFlag27,     {reserved}
   clipPix,          {specify to UpdateGWorld to update and clip }
                     { pixel image}
   stretchPix,       {specify to UpdateGWorld to update and }
                     { stretch or shrink pixel image}
   ditherPix,        {specify to UpdateGWorld to dither pixel }
                     { image}
   gwFlagErr,        {returned by UpdateGWorld if it failed}
);
```

Field descriptions

`pixPurge` If you specify this flag for the `flags` parameter of the `NewGWorld` function, `NewGWorld` (described on page 6-16) makes the base address for the offscreen pixel image purgeable.

`noNewDevice` If you specify this flag for the `flags` parameter of the `NewGWorld` function, `NewGWorld` does not create a new offscreen `GDevice` record; instead, `NewGWorld` uses either the `GDevice` record you specify or the `GDevice` record for a video card on the user's system.

`useTempMem` If you specify this in the `flags` parameter of the `NewGWorld` function, `NewGWorld` creates the base address for an offscreen pixel image in temporary memory. You generally should not use this flag. You should use temporary memory only for fleeting purposes and only with the `AllowPurgePixels` procedure (described on page 6-34) so that other applications can launch.

`keepLocal` If you specify this in the `flags` parameter of the `NewGWorld` function, `NewGWorld` creates a pixel image in Macintosh main memory where it cannot be cached to a graphics accelerator card.

`gWorldFlag4` Reserved.

`gWorldFlag5` Reserved.

`pixelsPurgeable`
If you specify this in the `state` parameter of the `SetPixelsState` procedure (described on page 6-37), `SetPixelsState` makes the base address for an offscreen pixel map purgeable. If you use the `SetPixelsState` procedure without passing it this flag, then `SetPixelsState` makes the base address for an offscreen pixel map unpurgeable. If the `GetPixelsState` function (described on page 6-36) returns this flag, then the base address for an offscreen pixel is purgeable.

`pixelsLocked`	If you specify this flag for the `state` parameter of the `SetPixelsState` procedure, `SetPixelsState` locks the base address for an offscreen pixel image. If you use the `SetPixelsState` procedure without passing it this flag, then `SetPixelsState` unlocks the offscreen pixel image. If the `GetPixelsState` function returns this flag, then the base address for an offscreen pixel is locked.
`gWorldFlag8`	Reserved.
`gWorldFlag9`	Reserved.
`gWorldFlag10`	Reserved.
`gWorldFlag11`	Reserved.
`gWorldFlag12`	Reserved.
`gWorldFlag13`	Reserved.
`gWorldFlag14`	Reserved.
`gWorldFlag15`	Reserved.
`mapPix`	If the `UpdateGWorld` function (described on page 6-23) returns this flag, then it remapped the colors in the offscreen pixel map to a new color table.
`newDepth`	If the `UpdateGWorld` function returns this flag, then it translated the offscreen pixel map to a different pixel depth.
`alignPix`	If the `UpdateGWorld` function returns this flag, then it realigned the offscreen pixel image to an onscreen window.
`newRowBytes`	If the `UpdateGWorld` function returns this flag, then it changed the `rowBytes` field of the `PixMap` record for the offscreen graphics world.
`reallocPix`	If the `UpdateGWorld` function returns this flag, then it reallocated the base address for the offscreen pixel image. Your application should then reconstruct the pixel image or draw directly in a window instead of preparing the image in an offscreen graphics world.
`gWorldFlag21`	Reserved.
`gWorldFlag22`	Reserved.
`gWorldFlag23`	Reserved.
`gWorldFlag24`	Reserved.
`gWorldFlag25`	Reserved.
`gWorldFlag26`	Reserved.
`gWorldFlag27`	Reserved.
`clipPix`	If the `UpdateGWorld` function returns this flag, then it clipped the pixel image.
`stretchPix`	If the `UpdateGWorld` function returns this flag, then it stretched or shrank the offscreen image.
`ditherPix`	If the `UpdateGWorld` function returns this flag, then it dithered the offscreen image.
`gwFlagErr`	If the `UpdateGWorld` function returns this flag, then it was unsuccessful and the offscreen graphics world was left unchanged.

Routines

This section describes routines for creating, altering, and disposing of offscreen graphics worlds; for saving and restoring offscreen graphics worlds; and for managing an offscreen graphics world's pixel map.

Creating, Altering, and Disposing of Offscreen Graphics Worlds

To create an offscreen graphics world, use the `NewGWorld` function. The `NewGWorld` function uses the `NewScreenBuffer` function to create and allocate memory for an offscreen pixel image; your application generally won't need to use `NewScreenBuffer`, but it is described here for completeness. The `NewGWorld` function similarly uses the `NewTempScreenBuffer` function to create and allocate temporary memory for an offscreen pixel image.

To change the pixel depth, boundary rectangle, or color table for an existing offscreen graphics world, use the `UpdateGWorld` function.

When you no longer need the pixel image associated with this offscreen graphics world, use the `DisposeGWorld` procedure to dispose of all the memory allocated for the offscreen graphics world. The `DisposeGWorld` procedure uses the `DisposeScreenBuffer` procedure when disposing of an offscreen graphics world; generally, your application won't need to use `DisposeScreenBuffer`.

Note
Before drawing into an offscreen graphics world, be sure to use the `SetGWorld` procedure (described on page 6-29) to make that offscreen world the current graphics port. In addition, before drawing into—or copying from—an offscreen pixel map, be sure to use the `LockPixels` function, which is described on page 6-32. ◆

NewGWorld

Use the `NewGWorld` function to create an offscreen graphics world.

```
FUNCTION NewGWorld (VAR offscreenGWorld: GWorldPtr;
                    pixelDepth: Integer; boundsRect: Rect;
                    cTable: CTabHandle; aGDevice: GDHandle;
                    flags: GWorldFlags): QDErr;
```

`offscreenGWorld`
A pointer to the offscreen graphics world created by this routine.

`pixelDepth`
The pixel depth of the offscreen world; possible depths are 1, 2, 4, 8, 16, and 32 bits per pixel. If you specify 0 in this parameter, you get the default behavior for the `NewGWorld` function—that is, it uses the pixel depth of the screen with the greatest pixel depth from among all screens whose boundary rectangles intersect the rectangle that you specify in the `boundsRect` parameter. If you specify 0 in this parameter, `NewGWorld` also uses the `GDevice` record from this device instead of creating a new `GDevice` record for the offscreen world. If you use `NewGWorld` on a computer that supports only basic QuickDraw, you may specify only 0 or 1 in this parameter.

`boundsRect`
The boundary rectangle and port rectangle for the offscreen pixel map. This becomes the boundary rectangle for the `GDevice` record, if `NewGWorld` creates one. If you specify 0 in the `pixelDepth` parameter, `NewGWorld` interprets the boundaries in global coordinates that it uses to determine which screens intersect the rectangle. (`NewGWorld` then uses the pixel depth, color table, and `GDevice` record from the screen with the greatest pixel depth from among all screens whose boundary rectangles intersect this rectangle.) Typically, your application supplies this parameter with the port rectangle for the onscreen window into which your application will copy the pixel image from this offscreen world.

`cTable` A handle to a `ColorTable` record. If you pass `NIL` in this parameter, `NewGWorld` uses the default color table for the pixel depth that you specify in the `pixelDepth` parameter. If you set the `pixelDepth` parameter to 0, `NewGWorld` ignores the `cTable` parameter and instead copies and uses the color table of the graphics device with the greatest pixel depth among all graphics devices whose boundary rectangles intersect the rectangle that you specify in the `boundsRect` parameter. If you use `NewGWorld` on a computer that supports only basic QuickDraw, you may specify only `NIL` in this parameter.

`aGDevice` A handle to a `GDevice` record that is used only when you specify the `noNewDevice` flag in the `flags` parameter, in which case `NewGWorld` attaches this `GDevice` record to the new offscreen graphics world. If you set the `pixelDepth` parameter to 0, or if you do not set the `noNewDevice` flag, `NewGWorld` ignores the `aGDevice` parameter, so you should set it to `NIL`. If you set the `pixelDepth` parameter to 0, `NewGWorld` uses the `GDevice` record for the graphics device with the greatest pixel depth among all graphics devices whose boundary rectangles intersect the rectangle that you specify in the `boundsRect` parameter. You should pass `NIL` in this parameter if the computer supports only basic QuickDraw. Generally, your application should never create `GDevice` records for offscreen graphics worlds.

`flags` Options available to your application. You can set a combination of the flags `pixPurge`, `noNewDevice`, `useTempMem`, and `keepLocal`. If you don't wish to use any of these flags, pass the empty set ([ ]) in your Pascal code or 0 in your C code in this parameter, in which case you get the default behavior for `NewGWorld`—that is, it creates an offscreen graphics world where the base address for the offscreen pixel image is unpurgeable, it uses an existing `GDevice` record (if you pass 0 in the `depth` parameter) or creates a new `GDevice` record, it uses memory in your application heap, and it allows graphics accelerators to cache the offscreen pixel image. The available flags are described here:

```
TYPE GWorldFlags =
SET OF (        {flags for only NewGWorld are listed here}
 pixPurge,      {make base address for offscreen pixel }
                { image purgeable}
 noNewDevice,   {do not create an offscreen GDevice }
                { record}
 useTempMem,    {create base address for offscreen pixel }
                { image in temporary memory}
 keepLocal,     {keep offscreen pixel image in main }
                { memory where it cannot be cached to }
                { a graphics accelerator card}
);
```

DESCRIPTION

The `NewGWorld` function creates an offscreen graphics world with the pixel depth you specify in the `pixelDepth` parameter, the boundary rectangle you specify in the `boundsRect` parameter, the color table you specify in the `cTable` parameter, and the options you specify in the `flags` parameter. The `NewGWorld` function returns a pointer to the new offscreen graphics world in the `offscreenGWorld` parameter. You use this pointer when referring to this new offscreen world in other routines described in this chapter.

Typically, you pass 0 in the `pixelDepth` parameter, a window's port rectangle in the `boundsRect` parameter, `NIL` in the `cTable` and `aGDevice` parameters, and—in the `flags` parameter—an empty set ([ ]) for Pascal code or 0 for C code. This provides your application with the default behavior of `NewGWorld`, and it supports computers running basic QuickDraw. This also allows QuickDraw to optimize the `CopyBits`, `CopyMask`, and `CopyDeepMask` procedures when your application copies the image in an offscreen graphics world into an onscreen graphics port.

The `NewGWorld` function allocates memory for an offscreen graphics port and its pixel map. On computers that support only basic QuickDraw, `NewGWorld` creates a 1-bit pixel map that your application can manipulate using other relevant routines described in this chapter. Your application can copy this 1-bit pixel map into basic graphics ports.

Unless you specify 0 in the `pixelDepth` parameter—or pass the `noNewDevice` flag in the `flags` parameter and supply a `GDevice` record in the `aGDevice` parameter—`NewGWorld` also allocates a new offscreen `GDevice` record.

When creating an image, your application can use the `NewGWorld` function to create an offscreen graphics world that is optimized for an image's characteristics—for example, its best pixel depth. After creating the image, your application can then use the `CopyBits`, `CopyMask`, or `CopyDeepMask` procedure to copy that image to an onscreen graphics port. Color QuickDraw automatically renders the image at the best available pixel depth for the screen. Creating an image in an offscreen graphics port and then copying it to the screen in this way prevents the visual choppiness that would otherwise occur if your application were to build a complex image directly onscreen.

The `NewGWorld` function initializes the offscreen graphics port by calling the `OpenCPort` function. The `NewGWorld` function sets the offscreen graphics port's visible region to a rectangular region coincident with its boundary rectangle. The `NewGWorld` function generates an inverse table with the Color Manager procedure `MakeITable`, unless one of the `GDevice` records for the screens has the same color table as the `GDevice` record for the offscreen world, in which case `NewGWorld` uses the inverse table from that `GDevice` record.

The address of the offscreen pixel image is not directly accessible from the `PixMap` record for the offscreen graphics world. However, you can use the `GetPixBaseAddr` function (described on page 6-38) to get a pointer to the beginning of the offscreen pixel image.

For purposes of estimating memory use, you can compute the size of the offscreen pixel image by using this formula:

```
rowBytes * (boundsRect.bottom – boundsRect.top)
```

In the `flags` parameter, you can specify several options that are defined by the `GWorldFlags` data type. If you don't wish to use any of these options, pass an empty set ([ ]) in the `flags` parameter for Pascal code or pass 0 here for C code.

- If you specify the `pixPurge` flag, `NewGWorld` stores the offscreen pixel image in a purgeable block of memory. In this case, before drawing to or from the offscreen pixel image, your application should call the `LockPixels` function (described on page 6-32) and ensure that it returns `TRUE`. If `LockPixels` returns `FALSE`, the memory for the pixel image has been purged, and your application should either call `UpdateGWorld` to reallocate it and then reconstruct the pixel image, or draw directly in a window instead of preparing the image in an offscreen graphics world. Never draw to or copy from an offscreen pixel image that has been purged without reallocating its memory and then reconstructing it.
- If you specify the `noNewDevice` flag, `NewGWorld` does not create a new offscreen `GDevice` record. Instead, it uses the `GDevice` record that you specify in the `aGDevice` parameter—and its associated pixel depth and color table—to create the offscreen graphics world. (If you set the `pixelDepth` parameter to 0, `NewGWorld` uses the `GDevice` record for the screen with the greatest pixel depth among all screens whose boundary rectangles intersect the rectangle that you specify in the `boundsRect` parameter—even if you specify the `noNewDevice` flag.) The

NewGWorld function keeps a reference to the GDevice record for the offscreen graphics world, and the SetGWorld procedure (described on page 6-29) uses that record to set the current graphics device.

- If you set the useTempMem flag, NewGWorld creates the base address for an offscreen pixel image in temporary memory. You generally would not use this flag, because you should use temporary memory only for fleeting purposes and only with the AllowPurgePixels procedure (described on page 6-34).
- If you specify the keepLocal flag, your offscreen pixel image is kept in Macintosh main memory and is not cached to a graphics accelerator card. Use this flag carefully, as it negates the advantages provided by any graphics acceleration card that might be present.

As its function result, NewGWorld returns one of three result codes.

SPECIAL CONSIDERATIONS

If you supply a handle to a ColorTable record in the cTable parameter, NewGWorld makes a copy of the record and stores its handle in the offscreen PixMap record. It is your application's responsibility to make sure that the ColorTable record you specify in the cTable parameter is valid for the offscreen graphics port's pixel depth.

If when using NewGWorld you specify a pixel depth, color table, or GDevice record that differs from those used by the window into which you copy your offscreen image, the CopyBits, CopyMask, and CopyDeepMask procedures require extra time to complete.

To use a custom color table in an offscreen graphics world, you need to create the associated offscreen GDevice record, because Color QuickDraw needs its inverse table.

The NewGWorld function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the NewGWorld function are

Trap macro	Selector
_QDExtensions	$00160000

RESULT CODES

noErr	0	No error
paramErr	–50	Illegal parameter
cDepthErr	–157	Invalid pixel depth

SEE ALSO

Listing 6-1 on page 6-5 and Listing 6-2 on page 6-10 illustrate how to use `NewGWorld` to create offscreen graphics worlds.

If your application needs to change the pixel depth, boundary rectangle, or color table for an offscreen graphics world, use the `UpdateGWorld` function, described on page 6-23.

NewScreenBuffer

The `NewGWorld` function uses the `NewScreenBuffer` function to create an offscreen `PixMap` record and allocate memory for the base address of its pixel image; applications generally don't need to use `NewScreenBuffer`.

```
FUNCTION NewScreenBuffer (globalRect: Rect;
                          purgeable: Boolean; VAR gdh: GDHandle;
                          VAR offscreenPixMap: PixMapHandle):
                          QDErr;
```

`globalRect`
The boundary rectangle, in global coordinates, for the offscreen pixel map.

`purgeable` A value of `TRUE` to make the memory block for the offscreen pixel map purgeable, or a value of `FALSE` to make it unpurgeable.

`gdh` The handle to the `GDevice` record for the graphics device with the greatest pixel depth among all graphics devices whose boundary rectangles intersect the rectangle specified in the `globalRect` parameter.

`offscreenPixMap`
A handle to the new offscreen `PixMap` record.

DESCRIPTION

The `NewScreenBuffer` function creates a new offscreen `PixMap` record, using the pixel depth and color table of the device whose `GDevice` record is returned in the `gdh` parameter. The `NewScreenBuffer` function returns a handle to the new offscreen pixel map in the `offscreenPixMap` parameter.

As its function result, `NewScreenBuffer` returns one of three result codes.

SPECIAL CONSIDERATIONS

The `NewScreenBuffer` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `NewScreenBuffer` function are

Trap macro	Selector
`_QDExtensions`	$000E0010

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Illegal parameter
`cNoMemErr`	–152	Failed to allocate memory for structures

NewTempScreenBuffer

The `NewGWorld` function uses the `NewTempScreenBuffer` function to create an offscreen `PixMap` record and allocate temporary memory for the base address of its pixel image; applications generally don't need to use `NewTempScreenBuffer`.

```
FUNCTION NewTempScreenBuffer (globalRect: Rect;
                              purgeable: Boolean;
                              VAR gdh: GDHandle;
                              VAR offscreenPixMap: PixMapHandle):
                              QDErr;
```

`globalRect`
The boundary rectangle, in global coordinates, for the offscreen pixel map.

`purgeable` A value of `TRUE` to make the memory block for the offscreen pixel map purgeable, or a value of `FALSE` to make it unpurgeable.

`gdh` The handle to the `GDevice` record for the graphics device with the greatest pixel depth among all graphics devices whose boundary rectangles intersect the rectangle specified in the `globalRect` parameter.

`offscreenPixMap`
A handle to the new offscreen `PixMap` record.

DESCRIPTION

The `NewTempScreenBuffer` function performs the same functions as `NewScreenBuffer` except that it creates the base address for the offscreen pixel image in temporary memory. When an application passes it the `useTempMem` flag, the `NewGWorld` function uses `NewTempScreenBuffer` instead of `NewScreenBuffer`.

SPECIAL CONSIDERATIONS

The `NewTempScreenBuffer` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `NewTempScreenBuffer` function are

Trap macro	Selector
`_QDExtensions`	$000E0015

UpdateGWorld

To change the pixel depth, boundary rectangle, or color table for an existing offscreen graphics world, use the `UpdateGWorld` function. You should call `UpdateGWorld` after every update event and whenever your windows move or change size.

```
FUNCTION UpdateGWorld (VAR offscreenGWorld: GWorldPtr;
                       pixelDepth: Integer; boundsRect: Rect;
                       cTable: CTabHandle; aGDevice: GDHandle;
                       flags: GWorldFlags): GWorldFlags;
```

`offscreenGWorld`
On input, a pointer to an existing offscreen graphics world; upon completion, the pointer to the updated offscreen graphics world.

`pixelDepth`
The pixel depth of the offscreen world; possible depths are 1, 2, 4, 8, 16, and 32 bits per pixel. If you specify 0 in this parameter, `UpdateGWorld` rescans the device list and uses the depth of the screen with the greatest pixel depth among all screens whose boundary rectangles intersect the rectangle that you specify in the `boundsRect` parameter. If you specify 0 in this parameter, `UpdateGWorld` also copies the `GDevice` record from this device to create an offscreen `GDevice` record. The `UpdateGWorld` function ignores the value you supply for this parameter if you specify a `GDevice` record in the `aGDevice` parameter.

`boundsRect`
The boundary rectangle and port rectangle for the offscreen pixel map. This also becomes the boundary rectangle for the `GDevice` record, if `NewGWorld` creates one. If you specify 0 in the `pixelDepth` parameter, `NewGWorld` interprets the boundaries in global coordinates, with which it determines which screens intersect the rectangle. (`NewGWorld` then uses the pixel depth, color table, and `GDevice` record from the screen with the greatest pixel depth from among all screens whose boundary rectangles intersect this rectangle.) Typically, your application supplies this parameter with the port rectangle for the onscreen window into which your application will copy the pixel image from this offscreen world.

`cTable` A handle to a `ColorTable` record. If you pass `NIL` in this parameter, `UpdateGWorld` uses the default color table for the pixel depth that you specify in the `pixelDepth` parameter; if you set the `pixelDepth` parameter to 0, `UpdateGWorld` copies and uses the color table of the graphics device with the greatest pixel depth among all graphics devices whose boundary rectangles intersect the rectangle that you specify in the `boundsRect` parameter. The `UpdateGWorld` function ignores the value you supply for this parameter if you specify a `GDevice` record in the `aGDevice` parameter.

`aGDevice` As an option, a handle to a `GDevice` record whose pixel depth and color table you want to use for the offscreen graphics world. To use the pixel depth and color table that you specify in the `pixelDepth` and `cTable` parameters, set this parameter to `NIL`.

`flags` Options available to your application. You can set a combination of the flags `keepLocal`, `clipPix`, `stretchPix`, and `ditherPix`. If you don't wish to use any of these flags, pass the empty set ([ ]) in this parameter for Pascal code or pass 0 here for C code. However, you should pass either `clipPix` or `stretchPix` to ensure that the pixel map is updated to reflect the new color table. The available flags are described here:

```
TYPE GWorldFlags =
SET OF (        {flags for UpdateGWorld are listed here}
 keepLocal,     {keep data structures in main memory}
 clipPix,       {update and clip pixel image to new }
                { boundary rectangle}
 stretchPix,    {update and stretch or shrink pixel }
                { image to the new boundary rectangle}
 ditherPix,     {include with clipPix or stretchPix }
                { flag to dither the pixel image}
);
```

DESCRIPTION

The `UpdateGWorld` function changes an offscreen graphics world to the specified pixel depth, rectangle, color table, and options that you supply in the `pixelDepth`, `boundsRect`, `cTable`, and `flags` parameters, respectively. In the `offscreenGWorld` parameter, pass the pointer returned to your application by the `NewGWorld` function when you created the offscreen graphics world.

If the `LockPixels` function (described on page 6-32) reports that the Memory Manager has purged the base address for the offscreen pixel image, you can use `UpdateGWorld` to reallocate its memory. Your application should then reconstruct the pixel image or draw directly in a window instead of preparing the image in an offscreen graphics world.

In the `flags` parameter, you can specify the `keepLocal` flag, which keeps the offscreen pixel image in Macintosh main memory or returns the image to main memory if it had been previously cached. If you use `UpdateGWorld` without passing it the `keepLocal` flag, you allow the offscreen pixel image to be cached on a graphics accelerator card if one is present.

As its function result, `UpdateGWorld` returns the `gwFlagErr` flag if `UpdateGWorld` was unsuccessful; in this case, the offscreen graphics world is left unchanged. You can use the `QDError` function, described in the chapter "Color QuickDraw," to help you determine why `UpdateGWorld` failed.

If `UpdateGWorld` is successful, it returns a combination of the following flags, which are defined by the `GWorldFlags` data type:

Flag	Meaning
`mapPix`	Color QuickDraw remapped the colors to a new color table.
`newDepth`	Color QuickDraw translated the pixel values in the offscreen pixel image to those for a different pixel depth.
`alignPix`	QuickDraw realigned the offscreen image to the window.
`newRowBytes`	QuickDraw changed the value of the `rowBytes` field in the `PixMap` record for the offscreen graphics world.
`reallocPix`	QuickDraw had to reallocate memory for the offscreen pixel image; your application should then reconstruct the pixel image, or draw directly in a window instead of preparing the image in an offscreen graphics world.
`clipPix`	QuickDraw clipped the pixel image.
`stretchPix`	QuickDraw stretched or shrank the offscreen image.
`ditherPix`	Color QuickDraw dithered the offscreen pixel image.

The `UpdateGWorld` function uses the following algorithm when updating the offscreen pixel image:

1. If the color table that you specify in the `cTable` parameter is different from the previous color table, or if the color table associated with the `GDevice` record that you specify in the `aGDevice` parameter is different, Color QuickDraw maps the pixel values in the offscreen pixel map to the new color table.
2. If the value you specify in the `pixelDepth` parameter differs from the previous pixel depth, Color QuickDraw translates the pixel values in the offscreen pixel image to those for the new pixel depth.
3. If the rectangle you specify in the `boundsRect` parameter differs from, but has the same size as, the previous boundary rectangle, QuickDraw realigns the pixel image to the screen for optimum performance for the `CopyBits` procedure.
4. If the rectangle you specify in the `boundsRect` parameter is smaller than the previous boundary rectangle and you specify the `clipPix` flag, the pixel image is clipped along the bottom and right edges.

5. If the rectangle you specify in the `boundsRect` parameter is bigger than the previous boundary rectangle and you specify the `clipPix` flag, the bottom and right edges of the pixel image are undefined.
6. If the rectangle you specify in the `boundsRect` parameter is smaller than the previous boundary rectangle and you specify the `stretchPix` flag, the pixel image is reduced to the new size.
7. If the rectangle you specify in the `boundsRect` parameter is bigger than the previous boundary rectangle and you specify the `stretchPix` flag, the pixel image is stretched to the new size.
8. If the Memory Manager purged the base address for the offscreen pixel image, `UpdateGWorld` reallocates the memory, but the pixel image is lost. You must reconstruct it.

SPECIAL CONSIDERATIONS

The `UpdateGWorld` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `UpdateGWorld` function are

Trap macro	Selector
`_QDExtensions`	$00160003

DisposeGWorld

Use the `DisposeGWorld` procedure to dispose of all the memory allocated for an offscreen graphics world.

```
PROCEDURE DisposeGWorld (offscreenGWorld: GWorldPtr);
```

`offscreenGWorld`
A pointer to an offscreen graphics world.

DESCRIPTION

The `DisposeGWorld` procedure disposes of all the memory allocated for the offscreen graphics world pointed to in the `offscreenGWorld` parameter, including its pixel map, color table, pixel image, and `GDevice` record (if one was created). In the `offscreenGWorld` parameter, pass the pointer returned to your application by the `NewGWorld` function when you created the offscreen graphics world.

Call `DisposeGWorld` only when your application no longer needs the pixel image associated with this offscreen graphics world. If this offscreen graphics world was the current device, the current device is reset to the device stored in the global variable `MainDevice`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DisposeGWorld` procedure are

Trap macro	Selector
`_QDExtensions`	$00040004

DisposeScreenBuffer

The `DisposeGWorld` procedure uses the `DisposeScreenBuffer` procedure when disposing of an offscreen graphics world; generally, applications do not need to use `DisposeScreenBuffer`.

```
PROCEDURE DisposeScreenBuffer (offscreenPixMap: PixMapHandle);
```

`offscreenPixMap`
A handle to an existing offscreen `PixMap` record.

DESCRIPTION

The `DisposeScreenBuffer` procedure disposes of the memory allocated for the base address of an offscreen pixel image.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DisposeScreenBuffer` procedure are

Trap macro	Selector
`_QDExtensions`	$00040011

Saving and Restoring Graphics Ports and Offscreen Graphics Worlds

To save the current graphics port (basic, color, or offscreen) and the current `GDevice` record, use the `GetGWorld` procedure. To change the current graphics port (basic, color, or offscreen), use the `SetGWorld` procedure; any drawing your application performs then occurs in this graphics port.

You can use the `GetGWorldDevice` function to obtain a handle to the `GDevice` record associated with an offscreen graphics world.

GetGWorld

To save the current graphics port (basic, color, or offscreen) and the current `GDevice` record, use the `GetGWorld` procedure.

```
PROCEDURE GetGWorld (VAR port: CGrafPtr; VAR gdh: GDHandle);
```

`port` A pointer to an offscreen graphics world, color graphics port, or basic graphics port, depending on which is the current port.

`gdh` A handle to the `GDevice` record for the current device.

DESCRIPTION

The `GetGWorld` procedure returns a pointer to the current graphics port in the `port` parameter. This parameter can return values of type `GrafPtr`, `CGrafPtr`, or `GWorldPtr`, depending on whether the current graphics port is a basic graphics port, color graphics port, or offscreen graphics world. The `GetGWorld` procedure returns a handle to the `GDevice` record for the current device in the `gdh` parameter.

After using `GetGWorld` to save a graphics port and a `GDevice` record, your application can later use the `SetGWorld` procedure, described next, to restore them.

SPECIAL CONSIDERATIONS

The `GetGWorld` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetGWorld` procedure are

Trap macro	Selector
`_QDExtensions`	$00080005

SEE ALSO

Listing 6-1 on page 6-5 and Listing 6-2 on page 6-10 illustrate how to use the `GetGWorld` procedure to save the current graphics port for an active window, the `SetGWorld` procedure to change the current graphics port to an offscreen graphics world, and then `SetGWorld` again to restore the active window as the current graphics port.

SetGWorld

To change the current graphics port (basic, color, or offscreen), use the `SetGWorld` procedure.

```
PROCEDURE SetGWorld (port: CGrafPtr; gdh: GDHandle);
```

`port` A pointer to an offscreen graphics world, color graphics port, or basic graphics port.

`gdh` A handle to a `GDevice` record. If you pass a pointer to an offscreen graphics world in the `port` parameter, set this parameter to `NIL`, because `SetGWorld` ignores this parameter and sets the current device to the device attached to the offscreen graphics world.

DESCRIPTION

The `SetGWorld` procedure sets the current graphics port to the one specified by the `port` parameter and—unless you set the current graphics port to be an offscreen graphics world—sets the current device to that specified by the `gdh` parameter.

In the `port` parameter, you can specify values of type `GrafPtr`, `CGrafPtr`, or `GWorldPtr`, depending on whether you want to set the current graphics port to be a basic graphics port, color graphics port, or offscreen graphics world. Any drawing your application performs then occurs in this graphics port.

SPECIAL CONSIDERATIONS

The `SetGWorld` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SetGWorld` procedure are

Trap macro	Selector
`_QDExtensions`	$00080006

SEE ALSO

Listing 6-1 on page 6-5 and Listing 6-2 on page 6-10 illustrate how to use the `GetGWorld` procedure to save the current graphics port for an active window, the `SetGWorld` procedure to change the current graphics port to an offscreen graphics world, and then `SetGWorld` again to restore the active window as the current graphics port.

GetGWorldDevice

Use the `GetGWorldDevice` function to obtain a handle to the `GDevice` record associated with an offscreen graphics world.

```
FUNCTION GetGWorldDevice (offscreenGWorld: GWorldPtr): GDHandle;
```

`offscreenGWorld`
A pointer to an offscreen graphics world. The pointer returned to your application by the `NewGWorld` function.

DESCRIPTION

The `GetGWorldDevice` function returns a handle to the `GDevice` record associated with the offscreen graphics world specified by the `offscreenGWorld` parameter. In this parameter, supply the pointer returned to your application by the `NewGWorld` function when you created the offscreen graphics world. If you created the offscreen world by specifying the `noNewDevice` flag, the `GDevice` record is for one of the screen devices or is the `GDevice` record that you specified to `NewGWorld` or `UpdateGWorld`.

If you point to a `GrafPort` or `CGrafPort` record in the `offscreenGWorld` parameter, `GetGWorldDevice` returns the current device.

SPECIAL CONSIDERATIONS

The `GetGWorldDevice` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetGWorldDevice` function are

Trap macro	Selector
`_QDExtensions`	$00040012

Managing an Offscreen Graphics World's Pixel Image

Use the `GetGWorldPixMap` function to obtain a handle to the `PixMap` record for an offscreen graphics world. You can then pass this handle, which is of data type `PixMapHandle`, in parameters to several routines that allow you to manage an offscreen graphics world's pixel image.

To prevent the base address for an offscreen pixel image from being moved (while you draw into or copy from its pixel map, for example), pass its handle to the `LockPixels` function. When you are finished drawing into or copying from an offscreen pixel map, pass its handle to the `UnlockPixels` procedure.

You can use the `AllowPurgePixels` procedure to make the base address for an offscreen pixel image purgeable. To prevent the Memory Manager from purging the base address for an offscreen pixel map, use the `NoPurgePixels` procedure.

To save the current information about the memory allocated for an offscreen pixel image, you can use the `GetPixelsState` function. To restore this state, you can use the `SetPixelsState` procedure.

You can use the `GetPixBaseAddr` function to obtain a pointer to the beginning of a pixel image in memory. You can use the `PixMap32Bit` function to determine whether a pixel map requires 32-bit addressing mode for access to its pixel image.

GetGWorldPixMap

Use the `GetGWorldPixMap` function to obtain the pixel map created for an offscreen graphics world.

```
FUNCTION GetGWorldPixMap (offscreenGWorld: GWorldPtr):
                          PixMapHandle;
```

`offscreenGWorld`
A pointer to an offscreen graphics world.

DESCRIPTION

The `GetGWorldPixMap` function returns a handle to the pixel map created for an offscreen graphics world. In the `offscreenGWorld` parameter, pass the pointer returned to your application by the `NewGWorld` function when you created the offscreen graphics world. Your application can, in turn, pass the handle returned by `GetGWorldPixMap` as a parameter to other QuickDraw routines that accept a handle to a pixel map.

On a system running only basic QuickDraw, the `GetGWorldPixMap` function returns the handle to a 1-bit pixel map that your application can supply as a parameter to the other routines related to offscreen graphics worlds. However, your application should not supply this handle to Color QuickDraw routines.

SPECIAL CONSIDERATIONS

To ensure compatibility on systems running basic QuickDraw instead of Color QuickDraw, use `GetGWorldPixMap` whenever you need to gain access to the bitmap created for a graphics world—that is, do *not* dereference the `GWorldPtr` record for that graphics world.

The `GetGWorldPixMap` function is not available in systems preceding System 7. You can make sure that the `GetGWorldPixMap` function is 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 `GetGWorldPixMap` is available.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetGWorldPixMap` function are

Trap macro	Selector
`_QDExtensions`	$00040017

SEE ALSO

The `Gestalt` function is described in the chapter "Gestalt Manager" of *Inside Macintosh: Operating System Utilities*.

LockPixels

To prevent the base address for an offscreen pixel image from being moved while you draw into or copy from its pixel map, use the `LockPixels` function.

```
FUNCTION LockPixels (pm: PixMapHandle): Boolean;
```

`pm` A handle to an offscreen pixel map. To get a handle to an offscreen pixel map, use the `GetGWorldPixMap` function, described on page 6-31.

DESCRIPTION

The `LockPixels` function prevents the base address for an offscreen pixel image from being moved. You must call `LockPixels` before drawing to or copying from an offscreen graphics world.

The `baseAddr` field of the `PixMap` record for an offscreen graphics world contains a handle instead of a pointer (which is what the `baseAddr` field for an onscreen pixel map contains). The `LockPixels` function dereferences the `PixMap` handle into a pointer. When you use the `UnlockPixels` procedure, which is described next, the handle is recovered.

If the base address for an offscreen pixel image hasn't been purged by the Memory Manager or is not purgeable, `LockPixels` returns `TRUE` as its function result, and your application can draw into or copy from the offscreen pixel map. However, if the base address for an offscreen pixel image has been purged, `LockPixels` returns `FALSE` to indicate that you can perform no drawing to or copying from the pixel map. At that point, your application should either call the `UpdateGWorld` function (described on

page 6-23) to reallocate the offscreen pixel image and then reconstruct it, or draw directly in a window instead of preparing the image in an offscreen graphics world.

As soon as you are finished drawing into and copying from the offscreen pixel image, you should call the `UnlockPixels` procedure.

SPECIAL CONSIDERATIONS

The `LockPixels` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LockPixels` function are

Trap macro	Selector
`_QDExtensions`	$00040001

SEE ALSO

Listing 6-1 on page 6-5 and Listing 6-2 on page 6-10 illustrate the use of this function. See *Inside Macintosh: Memory* for more information about memory management.

UnlockPixels

When you are finished drawing into and copying from an offscreen graphics world, use the `UnlockPixels` procedure.

```
PROCEDURE UnlockPixels (pm: PixMapHandle);
```

`pm` A handle to an offscreen pixel map. Pass the same handle that you passed previously to the `LockPixels` function.

DESCRIPTION

The `UnlockPixels` procedure allows the Memory Manager to move the base address for the offscreen pixel map that you specify in the `pm` parameter. To ensure the integrity of the data in a pixel image, call `LockPixels` (explained in the preceding section) before drawing into or copying from a pixel map; then, to prevent heap fragmentation, call `UnlockPixels` as soon as your application finishes drawing to and copying from the offscreen pixel map.

The `baseAddr` field of the `PixMap` record for an offscreen graphics world contains a handle instead of a pointer (which is what the `baseAddr` field for an onscreen pixel map contains). The `LockPixels` function dereferences the `PixMap` handle into a pointer. When you use the `UnlockPixels` procedure, the handle is recovered.

You don't need to call `UnlockPixels` if `LockPixels` returns `FALSE`, because `LockPixels` doesn't lock the memory for a pixel image if that memory has been purged. However, calling `UnlockPixels` on purged memory does no harm.

SPECIAL CONSIDERATIONS

The `UnlockPixels` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `UnlockPixels` procedure are

Trap macro	Selector
`_QDExtensions`	$00040002

AllowPurgePixels

You can use the `AllowPurgePixels` procedure to make the base address for an offscreen pixel image purgeable.

```
PROCEDURE AllowPurgePixels (pm: PixMapHandle);
```

`pm` A handle to an offscreen pixel map.

DESCRIPTION

The `AllowPurgePixels` procedure marks the base address for an offscreen pixel image as purgeable; this allows the Memory Manager to free the memory it occupies if available memory space becomes low. By default, `NewGWorld` creates an unpurgeable base address for an offscreen pixel image.

To get a handle to an offscreen pixel map, first use the `GetGWorldPixMap` function, described on page 6-31. Then supply this handle for the `pm` parameter of `AllowPurgePixels`.

Your application should call the `LockPixels` function (described on page 6-32) before drawing into or copying from an offscreen pixel map. If the Memory Manager has purged the base address for its pixel image, `LockPixels` returns `FALSE`. In that case either your application should use the `UpdateGWorld` function (described on page 6-23) to begin reconstructing the offscreen pixel image, or it should draw directly to an onscreen graphics port.

Only unlocked memory blocks can be made purgeable. If you use `LockPixels`, you must use the `UnlockPixels` procedure (explained in the preceding section) before calling `AllowPurgePixels`.

SPECIAL CONSIDERATIONS

The `AllowPurgePixels` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `AllowPurgePixels` procedure are

Trap macro	Selector
`_QDExtensions`	$0004000B

SEE ALSO

See *Inside Macintosh: Memory* for more information about memory management.

NoPurgePixels

To prevent the Memory Manager from purging the base address for an offscreen pixel image, use the `NoPurgePixels` procedure.

```
PROCEDURE NoPurgePixels (pm: PixMapHandle);
```

`pm` A handle to an offscreen pixel map.

DESCRIPTION

The `NoPurgePixels` procedure marks the base address for an offscreen pixel image as unpurgeable. To get a handle to an offscreen pixel map, use the `GetGWorldPixMap` function, described on page 6-31. Then supply this handle for the `pm` parameter of `NoPurgePixels`.

SPECIAL CONSIDERATIONS

The `NoPurgePixels` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `NoPurgePixels` procedure are

Trap macro	Selector
`_QDExtensions`	$0004000C

GetPixelsState

To save the current information about the memory allocated for an offscreen pixel image, you can use the `GetPixelsState` function.

```
FUNCTION GetPixelsState (pm: PixMapHandle): GWorldFlags;
```

`pm` A handle to an offscreen pixel map.

DESCRIPTION

The `GetPixelsState` function returns information about the memory allocated for the base address for an offscreen pixel image. This information can be either of the following flags defined by the `GWorldFlags` data type:

```
TYPE GWorldFlags =
SET OF (       {flags for GetPixelsState only are listed here}
   pixelsPurgeable,  {the base address for an offscreen pixel }
                     { image is purgeable}
   pixelsLocked,     {the offscreen pixel image is locked and }
                     { not purgeable}
);
```

If the `pixelsPurgeable` flag is not returned, then the base address for the offscreen pixel image is unpurgeable. If the `pixelsLocked` flag is not returned, then the base address for the offscreen pixel image is unlocked.

After using `GetPixelsState` to save this state information, your application can later use the `SetPixelsState` procedure, described next, to restore this state to the offscreen graphics world.

Specify a handle to a pixel map in the `pm` parameter. To get a handle to an offscreen pixel map, use the `GetGWorldPixMap` function, described on page 6-31.

SPECIAL CONSIDERATIONS

The `GetPixelsState` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetPixelsState` function are

Trap macro	Selector
`_QDExtensions`	$0004000D

SEE ALSO

After using `GetPixelsState` and before using `SetPixelsState`, your application can temporarily use the `AllowPurgePixels` procedure (described on page 6-34) to make the base address for an offscreen pixel image purgeable, the `NoPurgePixels` procedure (described on page 6-35) to make it unpurgeable, the `LockPixels` function (described on page 6-32) to prevent it from being moved, and the `UnlockPixels` procedure (described on page 6-33) to allow it to be moved.

SetPixelsState

To restore an offscreen pixel image to the state that you saved with the `GetPixelsState` function (explained in the preceding section), you can use the `SetPixelsState` procedure.

```
PROCEDURE SetPixelsState (pm: PixMapHandle; state: GWorldFlags);
```

`pm` A handle to an offscreen pixel map.

`state` Flags, which you usually save with the `GetPixelsState` function, defined by the `GWorldFlags` data type:

```
TYPE GWorldFlags =
SET OF ( {flags for SetPixelsState are listed here}
  pixelsPurgeable,  {make the base address for an }
                    { offscreen pixel image purgeable}
  pixelsLocked      {prevent the base address for an }
                    { offscreen pixel image from }
                    { being moved}
);
```

DESCRIPTION

The `SetPixelsState` procedure changes the state of the memory allocated for an offscreen pixel image to the state indicated by the flags specified in the `state` parameter, which you typically save using the `GetPixelsState` function.

Because only an unlocked memory block can be purged, `SetPixelsState` calls the `UnlockPixels` and `AllowPurgePixels` procedures (described on page 6-33 and page 6-34, respectively) if the `state` parameter specifies the `pixelsPurgeable` flag. If the `state` parameter does not specify the `pixelsPurgeable` flag, `SetPixelsState` makes the base address for the offscreen pixel image unpurgeable.

If the `state` parameter does not specify the `pixelsLocked` flag, `SetPixelsState` allows the base address for the offscreen pixel image to be moved.

SPECIAL CONSIDERATIONS

The `SetPixelsState` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SetPixelsState` procedure are

Trap macro	Selector
`_QDExtensions`	$0008000E

SEE ALSO

After using `GetPixelsState` and before using `SetPixelsState`, your application can temporarily alter the offscreen graphics world by using the `AllowPurgePixels` procedure (described on page 6-34) to temporarily mark the memory block for its offscreen pixel map as purgeable, the `NoPurgePixels` procedure (described on page 6-35) to make it unpurgeable, the `LockPixels` function (described on page 6-32) to prevent it from being moved, and the `UnlockPixels` procedure (described on page 6-33) to unlock it.

GetPixBaseAddr

You can use the `GetPixBaseAddr` function to obtain a pointer to an offscreen pixel map.

```
FUNCTION GetPixBaseAddr (pm: PixMapHandle): Ptr;
```

`pm` A handle to an offscreen pixel map. To get a handle to an offscreen pixel map, use the `GetGWorldPixMap` function, described on page 6-31.

DESCRIPTION

The `GetPixBaseAddr` function returns a 32-bit pointer to the beginning of a pixel image. The `baseAddr` field of the `PixMap` record for an offscreen graphics world contains a handle instead of a pointer, which is what the `baseAddr` field for an onscreen pixel map contains. You must use the `GetPixBaseAddr` function to obtain a pointer to the `PixMap` record for an offscreen graphics world.

Your application should never directly access the `baseAddr` field of the `PixMap` record for an offscreen graphics world; instead, your application should always use `GetPixBaseAddr`. If your application is using 24-bit mode, your application should then use the `PixMap32Bit` function (described next) to determine whether a pixel map requires 32-bit addressing mode for access to its pixel image.

If the offscreen buffer has been purged, `GetPixBaseAddr` returns `NIL`.

SPECIAL CONSIDERATIONS

Any QuickDraw routines that your application uses after calling `GetPixBaseAddr` may change the base address for the offscreen pixel image.

The `GetPixBaseAddr` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetPixBaseAddr` function are

Trap macro	Selector
`_QDExtensions`	$0004000F

SEE ALSO

See *Inside Macintosh: Memory* for information about determining addressing modes.

PixMap32Bit

You can use the `PixMap32Bit` function to determine whether a pixel map requires 32-bit addressing mode for access to its pixel image.

```
FUNCTION PixMap32Bit (pmHandle: PixMapHandle): Boolean;
```

`pmHandle` A handle to an offscreen pixel map.

DESCRIPTION

The `PixMap32Bit` function returns `TRUE` if a pixel map requires 32-bit addressing mode for access to its pixel image. If your application is in 24-bit mode, you must change to 32-bit mode.

To get a handle to an offscreen pixel map, first use the `GetGWorldPixMap` function, described on page 6-31. Then supply this handle for the `pm` parameter of `PixMap32Bit`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PixMap32Bit` function are

Trap macro	Selector
`_QDExtensions`	$00040016

SEE ALSO

See *Inside Macintosh: Memory* for information about determining and setting addressing modes.

Summary of Offscreen Graphics Worlds

Pascal Summary

Constants

```
CONST
   cDepthErr            = -157;  {invalid pixel depth}
   pixPurgeBit          = 0;     {set to to make base address for  }
                                 { offscreen pixel image purgeable}
   noNewDeviceBit       = 1;     {set to not create a new GDevice }
                                 { record for offscreen world}
   useTempMemBit        = 2;     {set to create base address for offscreen }
                                 { pixel image in temporary memory}
   keepLocalBit         = 3;     {set to keep offscreen pixel image in }
                                 { main memory}
   pixelsPurgeableBit   = 6;     {set to make base address for pixel image }
                                 { purgeable}
   pixelsLockedBit      = 7;     {set to lock base address for offscreen }
                                 { pixel image}
   mapPixBit            = 16;    {set by UpdateGWorld if it remapped }
                                 { colors to a new color table}
   newDepthBit          = 17;    {set by UpdateGWorld if it translated }
                                 { pixel map to a different pixel depth}
   alignPixBit          = 18;    {set by UpdateGWorld if it realigned }
                                 { pixel image to onscreen window}
   newRowBytesBit       = 19;    {set by UpdateGWorld if it changed }
                                 { rowBytes field of PixMap record}
   reallocPixBit        = 20;    {set by UpdateGWorld if it reallocated }
                                 { base address for offscreen pixel image}
   clipPixBit           = 28;    {set to clip pixel image}
   stretchPixBit        = 29;    {set to stretch or shrink pixel image}
   ditherPixBit         = 30;    {set to dither pixel image}
   gwFlagErrBit         = 31;    {set by UpdateGWorld if it failed}
```

Data Types

```
TYPE GWorldPtr = CGrafPtr;

TYPE GWorldFlags =
   SET OF (
      pixPurge,          {specify to NewGWorld to make base address for }
                         { offscreen pixel image purgeable}
      noNewDevice,       {specify to NewGWorld to not create a new GDevice }
                         { record for offscreen world}
      useTempMem,        {specify to NewGWorld to create base address for }
                         { offscreen pixel image in temporary memory}
      keepLocal,         {specify to NewGWorld to keep offscreen pixel image }
                         { in main memory}
      gWorldFlag4,       {reserved}
      gWorldFlag5,       {reserved}
      pixelsPurgeable,   {returned by GetPixelsState to indicate that base }
                         { address for offscreen pixel image is purgeable; }
                         { specify to SetPixelsState to make base address }
                         { for pixel image purgeable}
      pixelsLocked,      {returned by GetPixelsState to indicate that base }
                         { address for offscreen pixel image is locked; }
                         { specify to SetPixelsState to lock base address }
                         { for offscreen pixel image}
      gWorldFlag8,       {reserved}
      gWorldFlag9,       {reserved}
      gWorldFlag10,      {reserved}
      gWorldFlag11,      {reserved}
      gWorldFlag12,      {reserved}
      gWorldFlag13,      {reserved}
      gWorldFlag14,      {reserved}
      gWorldFlag15,      {reserved}
      mapPix,            {returned by UpdateGWorld if it remapped colors to }
                         { a new color table}
      newDepth,          {returned by UpdateGWorld if it translated pixel }
                         { map to a different pixel depth}
      alignPix,          {returned by UpdateGWorld if it realigned pixel }
                         { image to onscreen window}
      newRowBytes,       {returned by UpdateGWorld if it changed rowBytes }
                         { field of PixMap record}
      reallocPix,        {returned by UpdateGWorld if it reallocated }
                         { base address for offscreen pixel image}
      gWorldFlag21,      {reserved}
      gWorldFlag22,      {reserved}
```

```
      gWorldFlag23,     {reserved}
      gWorldFlag24,     {reserved}
      gWorldFlag25,     {reserved}
      gWorldFlag26,     {reserved}
      gWorldFlag27,     {reserved}
      clipPix,          {specify to UpdateGWorld to update and clip pixel }
                        { image}
      stretchPix,       {specify to UpdateGWorld to update and stretch or }
                        { shrink pixel image}
      ditherPix,        {specify to UpdateGWorld to dither pixel image}
      gwFlagErr,        {returned by UpdateGWorld if it failed}
   );
```

Routines

Creating, Altering, and Disposing of Offscreen Graphics Worlds

```
FUNCTION NewGWorld          (VAR offscreenGWorld: GWorldPtr;
                             pixelDepth: Integer; boundsRect: Rect;
                             cTable: CTabHandle; aGDevice: GDHandle;
                             flags: GWorldFlags): QDErr;
FUNCTION NewScreenBuffer    (globalRect: Rect;
                             purgeable: Boolean; VAR gdh: GDHandle;
                             VAR offscreenPixMap: PixMapHandle): QDErr;
FUNCTION NewTempScreenBuffer
                            (globalRect: Rect;
                             purgeable: Boolean;
                             VAR gdh: GDHandle;
                             VAR offscreenPixMap: PixMapHandle): QDErr;
FUNCTION UpdateGWorld       (VAR offscreenGWorld: GWorldPtr;
                             pixelDepth: Integer; boundsRect: Rect;
                             cTable: CTabHandle; aGDevice: GDHandle;
                             flags: GWorldFlags): GWorldFlags;
PROCEDURE DisposeGWorld     (offscreenGWorld: GWorldPtr);
PROCEDURE DisposeScreenBuffer
                            (offscreenPixMap: PixMapHandle);
```

Saving and Restoring Graphics Ports and Offscreen Graphics Worlds

```
PROCEDURE GetGWorld         (VAR port: CGrafPtr; VAR gdh: GDHandle);
PROCEDURE SetGWorld         (port: CGrafPtr; gdh: GDHandle);
FUNCTION GetGWorldDevice    (offscreenGWorld: GWorldPtr): GDHandle;
```

Managing an Offscreen Graphics World's Pixel Image

```
FUNCTION GetGWorldPixMap     (offscreenGWorld: GWorldPtr): PixMapHandle;
FUNCTION LockPixels          (pm: PixMapHandle): Boolean;
PROCEDURE UnlockPixels       (pm: PixMapHandle);
PROCEDURE AllowPurgePixels   (pm: PixMapHandle);
PROCEDURE NoPurgePixels      (pm: PixMapHandle);
FUNCTION GetPixelsState      (pm: PixMapHandle): GWorldFlags;
PROCEDURE SetPixelsState     (pm: PixMapHandle; state: GWorldFlags);
FUNCTION GetPixBaseAddr      (pm: PixMapHandle): Ptr;
FUNCTION PixMap32Bit         (pmHandle: PixMapHandle): Boolean;
```

C Summary

Constants

```
enum {                          /* bit assignments for GWorldFlags data type */
   pixPurgeBit          = 0,    /* set to to make base address for
                                   offscreen pixel image purgeable */
   noNewDeviceBit       = 1,    /* set to not create a new GDevice
                                   record for offscreen world */
   useTempMemBit        = 2,    /* set to create base address for offscreen
                                   pixel image in temporary memory */
   keepLocalBit         = 3,    /* set to keep offscreen pixel image in
                                   main memory */
   pixelsPurgeableBit   = 6,    /* set to make base address for pixel image
                                   purgeable */
   pixelsLockedBit      = 7,    /* set to lock base address for offscreen
                                   pixel image */
   mapPixBit            = 16,   /* set by UpdateGWorld if it remapped
                                   colors to a new color table */
   newDepthBit          = 17,   /* set by UpdateGWorld if it translated
                                   pixel map to a different pixel depth */
   alignPixBit          = 18,   /* set by UpdateGWorld if it realigned
                                   pixel image to onscreen window */
   newRowBytesBit       = 19,   /* set by UpdateGWorld if it changed
                                   rowBytes field of PixMap record */
   reallocPixBit        = 20,   /* set by UpdateGWorld if it reallocated
                                   base address for offscreen pixel image */
   clipPixBit           = 28,   /* set to update and clip pixel image */
```

```
   stretchPixBit        = 29, /* set to update and stretch or shrink pixel
                              image */
   ditherPixBit         = 30, /* set to dither pixel image */
   gwFlagErrBit         = 31  /* set by UpdateGWorld if it failed */
};

enum {                   /* constants for GWorldFlags data type */
   pixPurge          = 1 << pixPurgeBit,
   noNewDevice       = 1 << noNewDeviceBit,
   useTempMem        = 1 << useTempMemBit,
   keepLocal         = 1 << keepLocalBit,
   pixelsPurgeable   = 1 << pixelsPurgeableBit,
   pixelsLocked      = 1 << pixelsLockedBit,
   mapPix            = 1 << mapPixBit,
   newDepth          = 1 << newDepthBit,
   alignPix          = 1 << alignPixBit,
   newRowBytes       = 1 << newRowBytesBit,
   reallocPix        = 1 << reallocPixBit,
   clipPix           = 1 << clipPixBit,
   stretchPix        = 1 << stretchPixBit,
   ditherPix         = 1 << ditherPixBit,
   gwFlagErr         = 1 << gwFlagErrBit
};

enum {
   cDepthErr         = -157   /* invalid pixel depth */
};
```

Data Types

```
typedef CGrafPtr GWorldPtr;

typedef unsigned long GWorldFlags;
```

Functions

Creating, Altering, and Disposing of Offscreen Graphics Worlds

```
pascal QDErr NewGWorld      (GWorldPtr *offscreenGWorld, short PixelDepth,
                             const Rect *boundsRect, CTabHandle cTable,
                             GDHandle aGDevice, GWorldFlags flags);
pascal QDErr NewScreenBuffer
                            (const Rect *globalRect, Boolean purgeable,
                             GDHandle *gdh, PixMapHandle *offscreenPixMap);
pascal QDErr NewTempScreenBuffer
                            (const Rect *globalRect, Boolean purgeable,
                             GDHandle *gdh, PixMapHandle *offscreenPixMap);
pascal GWorldFlags UpdateGWorld
                            (GWorldPtr *offscreenGWorld, short pixelDepth,
                             const Rect *boundsRect, CTabHandle cTable,
                             GDHandle aGDevice, GWorldFlags flags);
pascal void DisposeGWorld   (GWorldPtr offscreenGWorld);
pascal void DisposeScreenBuffer
                            (PixMapHandle offscreenPixMap);
```

Saving and Restoring Graphics Ports and Offscreen Graphics Worlds

```
pascal void GetGWorld       (CGrafPtr *port, GDHandle *gdh);
pascal void SetGWorld       (CGrafPtr port, GDHandle gdh);
pascal GDHandle GetGWorldDevice
                            (GWorldPtr offscreenGWorld);
```

Managing an Offscreen Graphics World's Pixel Image

```
pascal PixMapHandle GetGWorldPixMap
                            (GWorldPtr offscreenGWorld);
pascal Boolean LockPixels   (PixMapHandle pm);
pascal void UnlockPixels    (PixMapHandle pm);
pascal void AllowPurgePixels
                            (PixMapHandle pm);
pascal void NoPurgePixels   (PixMapHandle pm);
pascal GWorldFlags GetPixelsState
                            (PixMapHandle pm);
pascal void SetPixelsState  (PixMapHandle pm, GWorldFlags state);
pascal Ptr GetPixBaseAddr   (PixMapHandle pm);
pascal Boolean PixMap32Bit  (PixMapHandle pmHandle);
```

Assembly-Language Summary

Trap Macros Requiring Routine Selectors

```
_QDExtensions
```

Selector	Routine
$00160000	`NewGWorld`
$00040001	`LockPixels`
$00040002	`UnlockPixels`
$00160003	`UpdateGWorld`
$00040004	`DisposeGWorld`
$00080005	`GetGWorld`
$00080006	`SetGWorld`
$0004000B	`AllowPurgePixels`
$0004000C	`NoPurgePixels`
$0004000D	`GetPixelsState`
$0008000E	`SetPixelsState`
$0004000F	`GetPixBaseAddr`
$000E0010	`NewScreenBuffer`
$00040011	`DisposeScreenBuffer`
$00040012	`GetGWorldDevice`
$000E0015	`NewTempScreenBuffer`
$00040016	`PixMap32Bit`
$00040017	`GetGWorldPixMap`

Result Codes

`noErr`	0	No error
`paramErr`	–50	Illegal parameter
`cNoMemErr`	–152	Failed to allocate memory for structures
`cDepthErr`	–157	Invalid pixel depth

Pictures

Contents

About Pictures 7-4
Picture Formats 7-5
Opcodes: Drawing Commands and Picture Comments 7-6
Color Pictures in Basic Graphics Ports 7-6
'PICT' Files, 'PICT' Resources, and the 'PICT' Scrap Format 7-7
The Picture Utilities 7-8
Using Pictures 7-8
Creating and Drawing Pictures 7-10
Opening and Drawing Pictures 7-13
Drawing a Picture Stored in a 'PICT' File 7-13
Drawing a Picture Stored in the Scrap 7-17
Defining a Destination Rectangle 7-18
Drawing a Picture Stored in a 'PICT' Resource 7-20
Saving Pictures 7-21
Gathering Picture Information 7-24
Pictures Reference 7-26
Data Structures 7-27
QuickDraw and Picture Utilities Routines 7-36
Creating and Disposing of Pictures 7-36
Drawing Pictures 7-43
Collecting Picture Information 7-46
Application-Defined Routines 7-61
Resources 7-67
The Picture Resource 7-67
The Color-Picking Method Resource 7-68
Summary of Pictures and the Picture Utilities 7-69
Pascal Summary 7-69
Constants 7-69
Data Types 7-69

Routines 7-72
Application-Defined Routines 7-73
C Summary 7-73
Constants 7-73
Data Types 7-74
Functions 7-76
Application-Defined Functions 7-77
Assembly-Language Summary 7-78
Data Structures 7-78
Trap Macros 7-80
Result Codes 7-80

Pictures

This chapter describes QuickDraw **pictures,** which are sequences of saved drawing commands. Pictures provide a common medium for the sharing of image data. Pictures make it easier for your application to draw complex images defined in other applications; pictures also make it easier for other applications to display images created with your application. Virtually all applications should support the creation and drawing of pictures. All applications that support cut and paste, for example, should be able to draw pictures copied by the user from the Clipboard.

Read this chapter to learn how to record QuickDraw drawing commands into a picture and how to draw the picture later by playing back these commands. You should also read this chapter to learn about the Picture Utilities, which allow your application to gather information about pictures—such as their colors, fonts, picture comments, and resolution. You can also use the Picture Utilities to gather information about the colors in pixel maps. Your application can use this information in conjunction with the Palette Manager, for example, to provide the best selection of colors for displaying a picture or other pixel image on an indexed device.

The `OpenCPicture` function, available on all Macintosh computers running System 7, allows your application to create pictures in the **extended version 2 picture format.** This format allows your application to specify resolutions when creating pictures.

Pictures can be created in color or black and white. Computers supporting only basic QuickDraw use black and white to display pictures created in color.

As described in this chapter, your application can use File Manager or Resource Manager routines to save or open pictures stored in files. See the chapter "File Manager" in *Inside Macintosh: Files* for more information about the File Manager; see the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information about the Resource Manager. To store or retrieve pictures in the scrap—for example, when the user copies from or pastes to the Clipboard—you must use Scrap Manager routines. See the chapter "Scrap Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information about the Scrap Manager.

You typically use the information gathered with the Picture Utilities in conjunction with other system software managers. You might use the Picture Utilities to determine what fonts are used in a picture, for example, and then use Font Manager routines to help you determine whether those fonts are available on the user's system. Or, you might use the Picture Utilities to determine the most-used colors in a picture, and then use the Palette Manager or ColorSync Utilities to provide sophisticated support for these colors. For more information about fonts, see the chapter "Font Manager" in *Inside Macintosh: Text*. The Palette Manager and the ColorSync Utilities are described in *Inside Macintosh: Advanced Color Imaging*.

You can also save and collect picture comments within your picture, as described in this chapter. Typically, however, your application uses picture comments to include special drawing commands for printers. Therefore, picture comments are described in greater detail in Appendix B, "Using Picture Comments for Printing," in this book.

About Pictures

QuickDraw provides a simple set of routines for recording a collection of its drawing commands and then playing the collection back later. Such a collection of drawing commands (as well as its resulting image) is called a *picture*. A replayed collection of drawing commands results in the picture shown in Figure 7-1.

Figure 7-1 A picture of a party hat



When you use the `OpenCPicture` function (or the `OpenPicture` function) to begin defining a picture, QuickDraw collects your subsequent drawing commands in a data structure of type `Picture`. You can define a picture by using any of the drawing routines described in this book—with the exception of the `CopyMask`, `CopyDeepMask`, `SeedFill`, `SeedCFill`, `CalcMask`, and `CalcCMask` routines.

By using the `DrawPicture` procedure, you can draw onscreen the picture defined by the instructions stored in the `Picture` record. Your application typically does not directly manipulate the information in this record. Instead, using a handle to a `Picture` record, your application passes this information to QuickDraw routines and Picture Utilities routines.

Note
The `OpenPicture` function, which is similar to the `OpenCPicture` function, was created for earlier versions of system software. Because of the support for higher resolutions provided by the `OpenCPicture` function, you should use `OpenCPicture` instead of `OpenPicture` to create a picture. ◆

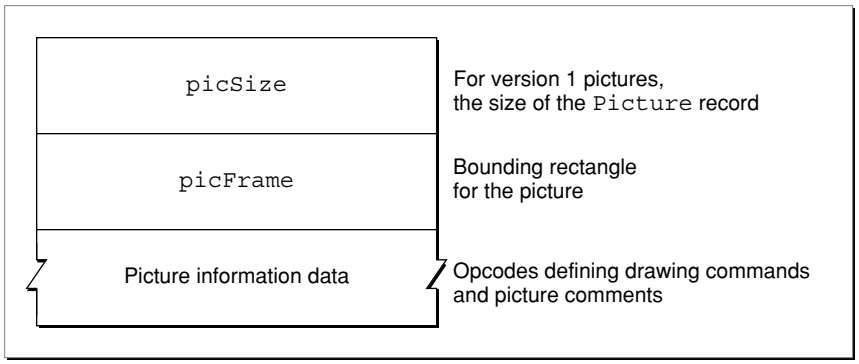
Picture Formats

Through QuickDraw's development, three different formats have evolved for the data contained in a `Picture` record. These formats are

- The extended version 2 format, which is created by the `OpenCPicture` function on all Macintosh computers running System 7, including those supporting only basic QuickDraw. This format permits your application to specify resolutions for pictures in color or black and white. Generally, your application should use the `OpenCPicture` function and create pictures in the extended version 2 format.
- The version 2 picture format, which is created by the `OpenPicture` function on machines with Color QuickDraw when the current graphics port is a color graphics port. Pictures created in this format support color drawing operations at 72 dpi.
- The original format, the version 1 picture format, which is created by the `OpenPicture` function on machines without Color QuickDraw or whenever the current graphics port is a basic graphics port. Pictures created in this format support only black-and-white drawing operations at 72 dpi.

The Pascal data structure for all picture formats is exactly the same. As shown in Figure 7-2, the `Picture` record begins with a `picSize` field and a `picFrame` field, followed by a variable amount of picture definition data.

Figure 7-2 The `Picture` record



To maintain compatibility with the version 1 picture format, the `picSize` field was not changed for the version 2 or extended version 2 picture formats. The information in this field is useful only for version 1 pictures, which cannot exceed 32 KB in size. Version 2 and extended version 2 pictures can be much larger than the 32 KB limit imposed by the 2-byte `picSize` field. You should use the Memory Manager function `GetHandleSize` to determine the size of a picture in memory, the File Manager function `PBGetFInfo` to determine the size of a picture in a file of type `'PICT'`, and the Resource Manager function `MaxSizeResource` to determine the size of a picture in a resource of type `'PICT'`. (See *Inside Macintosh: Memory*, *Inside Macintosh: Files*, and *Inside Macintosh: More Macintosh Toolbox* for more information about these functions.)

The `picFrame` field contains the bounding rectangle for the picture. The `DrawPicture` procedure uses this rectangle to scale the picture when you draw it into a differently sized rectangle.

Compact drawing instructions and picture comments constitute the rest of this record.

Opcodes: Drawing Commands and Picture Comments

Following the `picSize` and `picFrame` fields, a `Picture` record contains data in the form of **opcodes**, which are values that the `DrawPicture` procedure uses to determine what object to draw or what mode to change for subsequent drawing. Your application generally should not read or write these opcodes directly but should instead use the QuickDraw routines described in this chapter for generating and processing the opcodes. (For your application's debugging purposes, these opcodes are listed in Appendix A at the back of this book.)

In addition to compact QuickDraw drawing commands, opcodes can also specify picture comments. Created with the `PicComment` procedure, a **picture comment** contains data or commands for special processing by output devices, such as PostScript printers. If your application requires capability beyond that provided by QuickDraw drawing routines, the `PicComment` procedure allows your application to pass data or commands directly to the output device. For example, picture comments enable highly sophisticated drawing applications that process opcodes directly to reconstruct drawing instructions—such as rotating text—not found in QuickDraw. Picture comments are usually stored in the definition of a picture or are included in the code an application sends to a printer driver.

Unless your application creates highly sophisticated graphics, you typically use QuickDraw commands when drawing to the screen and use picture comments to include special drawing commands for printers only. For example, your application can use picture comments to specify commands—for rotating text and graphics and for drawing dashed lines, fractional line widths, and smoothed polygons—that are supported by some printers but are not accessible through standard QuickDraw calls. These picture comments are described in detail in Appendix B, "Using Picture Comments for Printing," in this book.

Color Pictures in Basic Graphics Ports

You can use Color QuickDraw drawing commands to create a color picture on a computer supporting Color QuickDraw. If the user were to cut the picture and paste it into an application that draws into a basic graphics port, the picture would lose some detail, but should be sufficient for most purposes. This is how basic QuickDraw in System 7 draws an extended version 2 or version 2 picture into a basic graphics port:

- QuickDraw maps foreground and background colors to those most closely approximated in basic QuickDraw's eight-color system.
- QuickDraw draws pixel patterns created with the `MakeRGBPat` procedure as bit patterns having approximately the same luminance as the pixel patterns.

- QuickDraw replaces other color patterns with the bit pattern contained in the `pat1Data` field of the `PixPat` record. (The `pat1Data` field is initialized to 50 percent gray if the pattern is created with the `NewPixPat` function; this field is initialized from a `'ppat'` resource if the pattern is retrieved with the `GetPixPat` function.)
- QuickDraw converts the pixel image to a bit image.
- QuickDraw ignores the values set by the `HiliteColor` and `OpColor` procedures, as well as any changes made to the `chExtra` and `pnLocHFrac` fields of the original `CGrafPort` record.

'PICT' Files, 'PICT' Resources, and the 'PICT' Scrap Format

QuickDraw provides routines for creating and drawing pictures; to read pictures from and to write pictures to disk, you use File Manager and Resource Manager routines. To read pictures from and write pictures to the scrap, you use Scrap Manager routines.

Files consist of two forks: a data fork and a resource fork. A **data fork** is the part of a file that contains data accessed using the File Manager. This data usually corresponds to data entered by the user. A **resource fork** is the part of a file that contains the file's resources, which contain data accessed using the Resource Manager. This data usually corresponds to data—such as menu, icon, and control definitions—created by the application developer, but it may also include data created by the user while the application is running.

A picture can be stored in the data fork of a file of type `'PICT'`. A picture can also be stored as a resource of type `'PICT'` in the resource fork of any file type.

Normally, an application sets the file type in the file's `FInfo` record when the application creates a new file; for example, the File Manager function `FSpCreate` takes a four-character file type—such as `'PICT'`—as a parameter. The data fork of a `'PICT'` file begins with a 512-byte header that applications can use for their own purposes. A `Picture` record follows this header. To read and write `'PICT'` files, your application should use File Manager routines.

You may find it useful to store pictures in the resource fork of your application or document file. For example, in response to the user choosing the About command in the Apple menu for your application, you might wish to display a window containing your company's logo. Or, if yours is a page-layout application, you might want to store all the images created by the user for a document as resources in the document file.

You can use high-level tools like the ResEdit resource editor, available from APDA, to create and store pictures as `'PICT'` resources for distribution with your files. To create `'PICT'` resources while your application is running, you should use Resource Manager routines. To retrieve a picture stored in a `'PICT'` resource, you can use the `GetPicture` function.

For each application, the Scrap Manager maintains a storage area to hold the last data cut or copied by the user. The area that is available to your application for this purpose is called the **scrap.** The scrap can reside either in memory or on disk. All applications that support copy-and-paste operations read data from and write data to the scrap. The

'PICT' scrap format is one of two standard scrap formats. (The other is 'TEXT'.) To support copy-and-paste operations, your application should use Scrap Manager routines to read and write data in 'PICT' scrap format.

The Picture Utilities

In addition to the QuickDraw routines for creating and drawing pictures, system software provides a group of routines called the **Picture Utilities** for examining the contents of pictures. You typically use the Picture Utilities before displaying a picture.

The Picture Utilities allow you to gather color, comment, font, resolution, and additional information about pictures. You might use the Picture Utilities, for example, to determine the 256 most-used colors in a picture, and then use the Palette Manager to make these colors available for the window in which your application needs to draw the picture.

You can also use the Picture Utilities to collect colors from pixel maps. You typically use this information in conjunction with the Palette Manager and the ColorSync Utilities to provide advanced color imaging features for your users. These features are described in *Inside Macintosh: Advanced Color Imaging*.

The Picture Utilities also collect information from black-and-white pictures and bitmaps. The Picture Utilities are supported in System 7 even by computers running only basic QuickDraw. 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.

Using Pictures

To create a picture, you should

- use the OpenCPicture function to create a Picture record and begin defining the picture
- issue QuickDraw drawing commands, which are collected in the Picture record
- use the PicComment procedure to include picture comments in the picture definition (optional)
- use the ClosePicture procedure to conclude the picture definition

To open an existing picture, you should

- use File Manager routines to get a picture stored in a 'PICT' file
- use the GetPicture function to get a picture stored in a 'PICT' resource
- use the Scrap Manager function GetScrap to get a picture stored in the scrap

To draw a picture, you should use the DrawPicture procedure.

To save a picture, you should

- use File Manager routines to save the picture in a `'PICT'` file
- use Resource Manager routines to save the picture in a `'PICT'` resource
- use the Scrap Manager function `PutScrap` to place the picture in the scrap

To conserve memory, you can spool large pictures to and from disk storage; you should

- write your own low-level procedures—using File Manager routines—that read and write temporary `'PICT'` files to disk
- use the `SetStdCProcs` procedure for a color graphics port (or the `SetStdProcs` procedure for a basic graphics port) and replace QuickDraw's standard low-level procedures `StdGetPic` and `StdPutPic` with your own procedures for reading and writing temporary `'PICT'` files to disk

To gather information about a single picture, pixel map, or bitmap, you should

- use the `GetPictInfo` function to get information about a picture, or use the `GetPixMapInfo` function to get information about a pixel map or bitmap
- use the `Palette` record or the `ColorTable` record, the handles of which are returned by these functions in a `PictInfo` record, to examine the colors collected from the picture, pixel map, or bitmap
- use the `FontSpec` record, the handle of which is returned by `GetPictInfo` in a `PictInfo` record, to examine the fonts contained in the picture
- use the `CommentSpec` record, the handle of which is returned by `GetPictInfo` in a `PictInfo` record, to examine the picture comments contained in the picture
- examine the rest of the fields of the `PictInfo` record for additional information—such as pixel depth or optimal resolution—about the picture, pixel map, or bitmap
- use the Memory Manager procedure `DisposeHandle` to release the memory occupied by the `PictInfo`, `FontSpec`, and `CommentSpec` records; use the Palette Manager procedure `DisposePalette` to release the memory occupied by a `Palette` record; and use the Color QuickDraw procedure `DisposeCTable` to release the memory occupied by a `ColorTable` record when you are finished with the information collected by the `GetPictInfo` function

To gather information about multiple pictures, pixel maps, and bitmaps, you should

- use the `NewPictInfo` function to begin collecting pictures, pixel maps, and bitmaps for your survey
- use the `RecordPictInfo` function to add the information for a picture to your survey
- use the `RecordPixMapInfo` function to add the information for a pixel map or bitmap to your survey
- use the `RetrievePictInfo` function to return the collected information in a `PictInfo` record
- use the `Palette` record or the `ColorTable` record, the handles of which are returned in the `PictInfo` record, to examine the colors collected from the pictures, pixel maps, and bitmaps

- use the `FontSpec` record, the handle of which is returned in the `PictInfo` record, to examine the fonts contained in the collected pictures
- use the `CommentSpec` record, the handle of which is returned in the `PictInfo` record, to examine the picture comments contained in the collected pictures
- examine the rest of the fields of the `PictInfo` record for additional information about the pictures, pixel maps, and bitmaps in your survey
- use the `DisposePictInfo` function to dispose of the private data structures allocated by the `NewPictInfo` function; use the Memory Manager procedure `DisposeHandle` to release the memory occupied by `PictInfo`, `FontSpec`, and `CommentSpec` records; use the Palette Manager procedure `DisposePalette` to release the memory occupied by a `Palette` record; and use the Color QuickDraw procedure `DisposeCTable` to release the memory occupied by a `ColorTable` record when you are finished with the information collected by `NewPictInfo`

When you are finished using a picture (such as when you close the window containing it), you should

- release the memory it occupies by calling the `KillPicture` procedure if the picture is not stored in a `'PICT'` resource
- release the memory it occupies by calling the Resource Manager procedure `ReleaseResource` if the picture is stored in a `'PICT'` resource

Before using the routines described in this chapter, you must use the `InitGraf` procedure, described in the chapter "Basic QuickDraw" in this book, to initialize QuickDraw. The routines in this chapter are available on all computers running System 7—including those supporting only basic QuickDraw. To test for the existence of System 7, use the `Gestalt` function with the `gestaltSystemVersion` selector. Test the low-order word in the `response` parameter; if the value is $0700 or greater, all of the routines in this chapter are supported.

Note
On computers running only basic QuickDraw, the Picture Utilities return `NIL` in place of handles to `Palette` and `ColorTable` records. ◆

Creating and Drawing Pictures

Use the `OpenCPicture` function to begin defining a picture. `OpenCPicture` collects your subsequent QuickDraw drawing commands in a new `Picture` record. To complete the collection of drawing and picture comment commands that define your picture, use the `ClosePicture` procedure.

Note
Operations with the following routines are not recorded in pictures: `CopyMask`, `CopyDeepMask`, `SeedFill`, `SeedCFill`, `CalcMask`, `CalcCMask`, and `PlotCIcon`. ◆

You pass information to `OpenCPicture` in the form of an `OpenCPicParams` record. This record provides a simple mechanism for specifying resolutions when creating images. For example, applications that create pictures from scanned images can specify resolutions higher than 72 dpi for these pictures in `OpenCPicParams` records.

Listing 7-1 shows an application-defined routine, `MyCreateAndDrawPict`, that begins creating a picture by assigning values to the fields of an `OpenCPicParams` record. In this example, the normal screen resolution of 72 dpi is specified as the picture's resolution. You also specify a rectangle for best displaying the picture at this resolution.

Listing 7-1 Creating and drawing a picture

```
FUNCTION MyCreateAndDrawPict(pFrame: Rect): PicHandle;
CONST
   cHRes = $00480000;   {for 72 dpi}
   cVRes = $00480000;   {for 72 dpi}
VAR
   myOpenCPicParams: OpenCPicParams;
   myPic:            PicHandle;
   trianglePoly:     PolyHandle;
BEGIN
   WITH myOpenCPicParams DO BEGIN
      srcRect := pFrame;   {best rectangle for displaying this picture}
      hRes := cHRes;       {horizontal resolution}
      vRes := cVRes;       {vertical resolution}
      version := - 2;      {always set this field to -2}
      reserved1 := 0;      {this field is unused}
      reserved2 := 0;      {this field is unused}
   END;
   myPic := OpenCPicture(myOpenCPicParams);  {start creating the picture}
   ClipRect(pFrame);                      {always set a valid clip region}
   FillRect(pFrame,dkGray);   {create a dark gray rectangle for background}
   FillOval(pFrame,ltGray);   {overlay the rectangle with a light gray oval}
   trianglePoly := OpenPoly;  {start creating a triangle}
   WITH pFrame DO BEGIN
      MoveTo(left,bottom);
      LineTo((right - left) DIV 2,top);
      LineTo(right,bottom);
      LineTo(left,bottom);
   END;
   ClosePoly;                 {finish the triangle}
   PaintPoly(trianglePoly);   {paint the triangle}
   KillPoly(trianglePoly);    {dispose of the memory for the triangle}
   ClosePicture;              {finish the picture}
```

```
   DrawPicture(myPic,pFrame);       {draw the picture}
   IF QDError <> noErr THEN
      ; {likely error is that there is insufficient memory}
   MyCreateAndDrawPict := myPic;
END;
```

After assigning values to the fields of an `OpenCPicParams` record, the `MyCreateAndDrawPict` routine passes this record to the `OpenCPicture` function.

IMPORTANT

Always use the `ClipRect` procedure to specify a clipping region appropriate for your picture before you call `OpenCPicture`. If you do not use `ClipRect` to specify a clipping region, `OpenCPicture` uses the clipping region specified in the current graphics port. If the clipping region is very large (as it is when a graphics port is initialized) and you scale the picture when drawing it, the clipping region can become invalid when `DrawPicture` scales the clipping region—in which case, your picture will not be drawn. On the other hand, if the graphics port specifies a small clipping region, part of your drawing may be clipped when you draw it. Setting a clipping region equal to the port rectangle of the current graphics port, as shown in Listing 7-1, always sets a valid clipping region. ▲

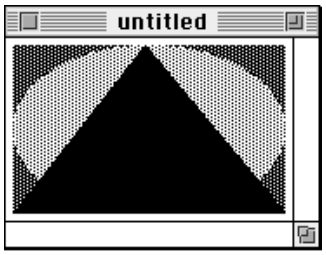
The `MyCreateAndDrawPict` routine uses QuickDraw commands to draw a filled rectangle, a filled oval, and a black triangle. These commands are stored in the `Picture` record.

Note

If there is insufficient memory to draw a picture in Color QuickDraw, the `QDError` function (described in the chapter "Color QuickDraw" in this book) returns the result code `noMemForPictPlaybackErr`. ◆

The `MyCreateAndDrawPict` routine concludes the picture definition by using the `ClosePicture` procedure. By passing to the `DrawPicture` procedure the handle to the newly defined picture, `MyCreateAndDrawPict` replays in the current graphics port the drawing commands stored in the `Picture` record. Figure 7-3 shows the resulting figure.

Figure 7-3 A simple picture



Note
After using `DrawPicture` to draw a picture, your application can use the Window Manager procedure `SetWindowPic` to save a handle to the picture in the window record. When the window's content region must be updated, the Window Manager draws this picture, or only a part of it as necessary, instead of generating an update event. Another Window Manager routine, the `GetWindowPic` function, allows your application to retrieve the picture handle that you store using `SetWindowPic`. When you use the Window Manager procedure `DisposeWindow` to close a window, `DisposeWindow` automatically calls the `KillPicture` procedure to release the memory allocated to a picture referenced in the window record. These routines and the window record are described in the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*. ◆

Opening and Drawing Pictures

Using File Manager routines, your application can retrieve pictures saved in `'PICT'` files; using the `GetPicture` function, your application can retrieve pictures saved in the resource forks of other file types; and using the Scrap Manager function `GetScrap`, your application can retrieve pictures stored in the scrap.

Drawing a Picture Stored in a 'PICT' File

Listing 7-2 illustrates an application-defined routine, called `MyDrawFilePicture`, that uses File Manager routines to retrieve a picture saved in a `'PICT'` file. The `MyDrawFilePicture` routine takes a file reference number as a parameter.

Listing 7-2 Opening and drawing a picture from disk

```
FUNCTION MyDrawFilePicture(pictFileRefNum: Integer; destRect: Rect): OSErr;
CONST
   cPicFileHeaderSize = 512;
VAR
   myPic:      PicHandle;
   dataLength: LongInt;
   err:        OSErr;
BEGIN    {This listing assumes the current graphics port is set.}
   err := GetEOF(pictFileRefNum,dataLength);    {get file length}
   IF err = noErr THEN BEGIN
      err := SetFPos(pictFileRefNum,fsFromStart,
                     cPicFileHeaderSize); {move past the 512-byte 'PICT' }
                                          { file header}
      dataLength := dataLength - cPicFileHeaderSize;  {remove 512-byte }
                                    { 'PICT' file header from file length}
      myPic := PicHandle(NewHandle(dataLength)); {allocate picture handle}
```

```
      IF (err = noErr) & (myPic <> NIL) THEN BEGIN
         HLock(Handle(myPic));   {lock picture handle before using FSRead}
         err := FSRead(pictFileRefNum,dataLength,Ptr(myPic^)); {read file}
         HUnlock(Handle(myPic)); {unlock picture handle after using FSRead}
         MyAdjustDestRect(myPic,destRect);   {see Listing 7-7 on page 7-18}
         DrawPicture(myPic,destRect);
         IF QDError <> noErr THEN
            ; {likely error is that there is insufficient memory}
         KillPicture(myPic);
      END;
   END;
   MyDrawFilePicture := err;
END;
```

In code not shown in Listing 7-2, this application uses the File Manager procedure `StandardGetFile` to display a dialog box that asks the user for the name of a `'PICT'` file; using the file system specification record returned by `StandardGetFile`, the application calls the File Manager function `FSpOpenDF` to return a file reference number for the file. The application then passes this file reference number to `MyDrawFilePicture`.

Because every `'PICT'` file contains a 512-byte header for application-specific use, `MyDrawFilePicture` uses the File Manager function `SetFPos` to skip past this header information. The `MyDrawFilePicture` function then uses the File Manager function `FSRead` to read the file's remaining bytes—those of the `Picture` record—into memory.

The `MyDrawFilePicture` function creates a handle for the buffer into which the `Picture` record is read. Passing this handle to the `DrawPicture` procedure, `MyDrawFilePicture` is able to replay onscreen the commands stored in the `Picture` record.

For large `'PICT'` files, it is useful to spool the picture data from disk as necessary instead of reading all of it directly into memory. In low-memory conditions, for example, your application might find it useful to create a temporary file on disk for storing drawing instructions; your application can read this information as necessary. The application-defined routine `MyReplaceGetPic` shown in Listing 7-3 replaces the `getPicProc` field of the current graphics port's `CQDProcs` record with an application-defined low-level routine, called `MyFileGetPic`. While QuickDraw's standard `StdGetPic` procedure reads picture data from memory, `MyFileGetPic` reads the picture data from disk. (Listing 7-10 on page 7-22 shows how to replace QuickDraw's standard `StdPutPic` procedure with one that writes data to a file so that your application can spool a large picture to disk.)

Listing 7-3 Replacing QuickDraw's standard low-level picture-reading routine

```
FUNCTION MyReplaceGetPic: QDProcsPtr;
VAR
   currPort:      GrafPtr;
   customProcs:   QDProcs;
   customCProcs:  CQDProcs;
   savedProcs:    QDProcsPtr;
BEGIN
   GetPort(currPort);
   savedProcs := currPort^.grafProcs;  {save current CQDProcs }
                                       { or QDProcs record}
   IF MyIsColorPort(currPort) THEN     {this is a color graphics port}
   BEGIN
      SetStdCProcs(customCProcs);   {create new CQDProcs record containing }
                                    { standard Color QuickDraw low-level }
                                    { routines}
      customCProcs.getPicProc := @MyFileGetPic; {replace StdGetPic with }
                                                { address of custom }
                                                { low-level routine }
                                                { shown in Listing 7-5}
      currPort^.grafProcs := @customCProcs;  {replace current CQDProcs }
                                             { record}
   END
   ELSE
   BEGIN                            {this is a basic graphics port}
      SetStdProcs(customProcs);     {create new QDProcs record containing }
                                    { standard basic QuickDraw low-level }
                                    { routines}
      customProcs.getPicProc := @MyFileGetPic;  {replace StdGetPic with }
                                                { address of custom }
                                                { low-level routine }
                                                { shown in Listing 7-5}
      currPort^.grafProcs := @customProcs;   {replace current QDProcs record}
   END;
   MyReplaceGetPic := savedProcs;
END;
```

Listing 7-4 shows the application-defined procedure `MyIsColorPort`, which `MyReplaceGetPic` calls to determine whether to replace the low-level picture-reading routine for a color graphics port or a basic graphics port.

Listing 7-4 Determining whether a graphics port is color or basic

```
FUNCTION MyIsColorPort(aPort: GrafPtr): Boolean;
BEGIN
   MyIsColorPort := (aPort^.portBits.rowBytes < 0)
END;
```

Listing 7-5 shows the application-defined procedure `MyFileGetPic`, which uses the File Manager function `FSRead` to read the file with the file reference number assigned to the application-defined global variable `gPictFileRefNum`.

Listing 7-5 A custom low-level procedure for spooling a picture from disk

```
PROCEDURE MyFileGetPic (dataPtr: Ptr; byteCount: Integer);
VAR
   longCount:  LongInt;
   myErr:      OSErr;
BEGIN
   longCount := byteCount;
   myErr := FSRead(gPictFileRefNum, longCount, dataPtr);
END;
```

Your application does not keep track of where `FSRead` stops or resumes reading a file. After reading a portion of a file, `FSRead` automatically handles where to begin reading next. See *Inside Macintosh: Files* for more information about using `FSRead` and other File Manager routines to retrieve data stored in files.

Drawing a Picture Stored in the Scrap

As described in the chapter "Scrap Manager" in *Inside Macintosh: More Macintosh Toolbox*, your application can use the Scrap Manager to copy and paste data within a document created by your application, among different documents created by your application, and among documents created by your application and documents created by other applications. The two standard scrap formats that all Macintosh applications should support are `'PICT'` and `'TEXT'`.

Listing 7-6 illustrates the application-defined routine `MyPastePict`, which retrieves a picture stored on the scrap. For example, a user may have copied to the Clipboard a picture created in another application and then pasted the picture into the application that defines `MyPastePict`. The `MyPastePict` procedure uses the Scrap Manager procedure `GetScrap` to get a handle to the data stored on the scrap; `MyPastePict` then coerces this handle to one of type `PicHandle`, which it can pass to the `DrawPicture` procedure in order to replay the drawing commands stored in the scrap.

Listing 7-6 Pasting in a picture from the scrap

```
PROCEDURE MyPastePict(destRect: Rect);
VAR
   myPic:      PicHandle;
   dataLength: LongInt;
   dontCare:   LongInt;
BEGIN
   myPic := PicHandle(NewHandle(0));   {allocate a handle for the picture}
   dataLength :=
            GetScrap(Handle(myPic),'PICT',dontCare);  {get picture in scrap}
   IF dataLength > 0 THEN {ensure there is PICT data}
   BEGIN
      MyAdjustDestRect(myPic,destRect);   {shown in Listing 7-7}
      DrawPicture(myPic,destRect);
      IF QDError <> noErr THEN
         ; {likely error is that there is insufficient memory}
   END
   ELSE
   ; {handle error for len < or = 0 here}
END;
```

Defining a Destination Rectangle

In addition to taking a handle to a picture as one parameter, `DrawPicture` also expects a destination rectangle as another parameter. You should specify this destination rectangle in coordinates local to the current graphics port. The `DrawPicture` procedure shrinks or stretches the picture as necessary to make it fit into this rectangle.

Listing 7-7 shows an application-defined routine called `MyAdjustDestRect` that centers the picture inside a destination rectangle, which is passed to `DrawPicture` when it's time to draw the picture. (`MyAdjustDestRect` first ensures that the picture fits inside the destination rectangle by scaling the picture if necessary.)

Listing 7-7 Adjusting the destination rectangle for a picture

```
PROCEDURE MyAdjustDestRect(aPict: PicHandle; VAR destRect: Rect);
VAR
   r:                      Rect;
   width, height:          Integer;
   scale, scaleH, scaleV:  Fixed;
BEGIN
   WITH destRect DO BEGIN  {determine width and height of destination rect}
      width := right - left;
      height := bottom - top;
   END;
   r := aPict^^.picFrame;        {get the bounding rectangle of the picture}
   OffsetRect(r, - r.left, - r.top);   {ensure upper-left corner is (0,0)}
   scale := Long2Fix(1);
   scaleH := FixRatio(width,r.right);     {get horizontal and vertical }
   scaleV := FixRatio(height,r.bottom);   { ratios of destination rectangle }
                                          { to bounding rectangle of picture}
   IF scaleH < scale THEN scale := scaleH;   {if bounding rect of picture }
   IF scaleV < scale THEN scale := scaleV;   { is greater than destination }
   IF scale <> Long2Fix(1) THEN              { rect, get scaling factors}
   BEGIN       {scale picture to fit inside destination rectangle}
      r.right := Fix2Long(FixMul(scale,Long2Fix(r.right)));
      r.bottom := Fix2Long(FixMul(scale,Long2Fix(r.bottom)));
    END;
   {next line centers the picture within the destination rectangle}
   OffsetRect(r,(width - r.right) DIV 2,(height - r.bottom) DIV 2);
   destRect := r;
END;
```

The application calling `MyAdjustDestRect` begins defining a destination rectangle by determining a target area within a window—perhaps the entire content area of a window, or perhaps an area selected by the user within a window. The application passes this rectangle to `MyAdjustDestRect`.

A bounding rectangle is stored in the `picFrame` field of the `Picture` record for every picture. The `MyAdjustDestRect` routine uses the boundaries for the picture to determine whether the picture fits within the destination rectangle. If the picture is larger than the destination rectangle, `MyAdjustDestRect` scales the picture to make it fit the destination rectangle.

The `MyAdjustDestRect` routine then centers the picture within the destination rectangle. Finally, `MyAdjustDestRect` assigns the boundary rectangle of the centered picture to be the new destination rectangle. By returning a destination rectangle whose dimensions are identical to those of the bounding rectangle for the picture, `MyAdjustDestRect` assures that the picture is not stretched when drawn into its window.

To display a picture at a resolution other than the one at which it was created, your application should compute an appropriate destination rectangle by scaling its width and height by the following factor:

scale factor = destination resolution / source resolution

For example, if a picture was created at 300 dpi and you want to display it at 75 dpi, then your application should compute the destination rectangle width and height as 1/4 of those of the picture's bounding rectangle. Your application can use the `GetPictInfo` function (described on page 7-47) to gather information about a picture. The `PictInfo` record (described on page 7-32) returned by `GetPictInfo` returns the picture's resolution in its `hRes` and `vRes` fields. The `sourceRect` field contains the bounding rectangle for displaying the image at its optimal resolution.

Drawing a Picture Stored in a 'PICT' Resource

To retrieve a picture stored in a `'PICT'` resource, specify its resource ID to the `GetPicture` function, which returns a handle to the picture. Listing 7-8 illustrates an application-defined routine, called `MyDrawResPICT`, that retrieves and draws a picture stored as a resource.

Listing 7-8 Drawing a picture stored in a resource file

```
PROCEDURE MyDrawResPICT(destRect: Rect; resID: Integer);
VAR
   myPic:   PicHandle;
BEGIN
   myPic := GetPicture(resID);   {get the picture from the resource fork}
   IF myPic <> NIL THEN BEGIN
      MyAdjustDestRect(myPic,destRect);{see Listing 7-7 on page 7-18}
      DrawPicture(myPic,destRect);
      IF QDError <> noErr THEN
         ; {likely error is that there is insufficient memory}
   END
   ELSE
   ; {handle the error here}
END;
```

When you are finished using a picture stored as a `'PICT'` resource, you should use the Resource Manager procedure `ReleaseResource` instead of the QuickDraw procedure `KillResource` to release its memory.

IMPORTANT

If you retrieve a picture stored in a `'PICT'` resource and pass its handle to the Window Manager procedure `SetWindowPic`, the Window Manager procedures `DisposeWindow` and `CloseWindow` do not delete it; instead, you must call `ReleaseResource` before calling `DisposeWindow` or `CloseWindow`. ▲

Saving Pictures

After creating or changing pictures, your application should allow the user to save them. To save a picture in a `'PICT'` file, you should use File Manager routines, such as `FSpCreate`, `FSpOpenDF`, `FSWrite`, and `FSClose`. The use of these routines is illustrated in Listing 7-9, and they are described in detail in the chapter "File Manager" in *Inside Macintosh: Files*. Remember that the first 512 bytes of a `'PICT'` file are reserved for your application's own purposes. As shown in Listing 7-9, your application should store the data (that is, the `Picture` record) after this 512-byte header.

Listing 7-9 Saving a picture as a `'PICT'` file

```
FUNCTION DoSavePICTAsCmd(picH: PicHandle): OSErr;
LABEL 8,9;
VAR
   myReply:                      StandardFileReply;
   err, ignore:                  OSErr;
   pictFileRefNum:               Integer;
   dataLength, zeroData, count:  LongInt;
BEGIN                {display the default Save dialog box}
   StandardPutFile('Save picture as:','untitled',myReply);
   err := noErr;  {return noErr if the user cancels}
   IF myReply.sfGood THEN
   BEGIN
      IF NOT myReply.sfReplacing THEN  {create the file if it doesn't exist}
         err := FSpCreate(myReply.sfFile,'WAVE','PICT',smSystemScript);
      IF err <> noErr THEN GOTO 9;
      err := FSpOpenDF(myReply.sfFile,fsRdWrPerm,pictFileRefNum); {open file}
      IF err <> noErr THEN GOTO 8;
      zeroData := 0;
      dataLength := 4;
      FOR count := 1 TO 512 DIV dataLength DO   {write the PICT file header}
         err := FSWrite(pictFileRefNum,dataLength,
                        @zeroData); {for this app, put 0's in header}
      IF err <> noErr THEN GOTO 8;
      dataLength := GetHandleSize(Handle(picH));
      HLock(Handle(picH)); {lock picture handle before writing data}
```

```
      err := FSWrite(pictFileRefNum,dataLength,Ptr(picH^)); {write picture }
                                                            { data to file}
       HUnlock(Handle(picH)); {unlock picture handle after writing data}
   END;
   8:
      ignore := FSClose(pictFileRefNum);  {close the file}
   9:
      DoSavePICTAsCmd := err;
END;
```

To save a picture in a `'PICT'` resource, you should use Resource Manager routines, such as `FSpOpenResFile` (to open your application's resource fork), `ChangedResource` (to change an existing `'PICT'` resource), `AddResource` (to add a new `'PICT'` resource), `WriteResource` (to write the data to the resource), and `CloseResFile` and `ReleaseResource` (to conclude saving the resource). These routines are described in the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*.

To place a picture in the scrap—for example, in response to the user choosing the Copy command to copy a picture to the Clipboard—use the Scrap Manager function `PutScrap`, which is described in the chapter "Scrap Manager" in *Inside Macintosh: More Macintosh Toolbox*.

For large `'PICT'` files, it is useful to spool the picture data to disk instead of writing it all directly into memory. In low-memory conditions, for example, your application might find it useful to create a temporary file on disk for storing drawing instructions; your application can read this information as necessary. The application-defined routine `MyReplacePutPic` shown in Listing 7-10 replaces the `putPicProc` field of the current graphics port's `CQDProcs` record with an application-defined low-level routine, called `MyFilePutPic`. While QuickDraw's standard `StdPutPic` procedure writes picture data to memory, `MyFilePutPic` writes the picture data to disk. (Listing 7-3 on page 7-15 shows how to replace QuickDraw's standard `StdGetPic` procedure with one that reads data from a spool file.)

Listing 7-10 Replacing QuickDraw's standard low-level picture-writing routine

```
FUNCTION MyReplacePutPic: QDProcsPtr;
VAR
   currPort:      GrafPtr;
   customProcs:   QDProcs;
   customCProcs:  CQDProcs;
   savedProcs:    QDProcsPtr;
BEGIN
   GetPort(currPort);
   savedProcs := currPort^.grafProcs;  {save QDProcs or CQDProcs record }
                                       { for current graphics port}
   IF MyIsColorPort(currPort) THEN  {see Listing 7-4 on page 7-16}
```

```
   BEGIN
      SetStdCProcs(customCProcs);   {create new CQDProcs record containing }
                                    { standard Color QuickDraw low-level }
                                    { routines}

      customCProcs.putPicProc := @MyFilePutPic; {replace StdPutPic with }
                                                { address of custom }
                                                { low-level routine }
                                                { shown in Listing 7-11}
      currPort^.grafProcs := @customCProcs;  {replace current CQDProcs}
   END
   ELSE
   BEGIN       {perform similar work for a basic graphics port}
      SetStdProcs(customProcs);
      customProcs.putPicProc := @MyFilePutPic;
      currPort^.grafProcs := @customProcs;
   END;
   gPictureSize := 0;   {track the picture size}
   gSpoolPicture := PicHandle(NewHandle(0));
   MyReplacePutPic := savedProcs;         {return saved CQDProcs or QDProcs }
                                    { record for restoring at a later time}
END;
```

Listing 7-11 shows `MyFilePutPic`, which uses the File Manager function `FSWrite` to write picture data to the file with the file reference number assigned to the application-defined global variable `gPictFileRefNum`. Your application does not keep track of where `FSWrite` stops or resumes writing a file. After writing a portion of a file, `FSWrite` automatically handles where to begin writing next.

Listing 7-11 A custom low-level routine for spooling a picture to disk

```
PROCEDURE MyFilePutPic (dataPtr: Ptr; byteCount: Integer);
VAR
   dataLength: LongInt;
   myErr: OSErr;
BEGIN
   dataLength := byteCount;
   gPictureSize := gPictureSize + byteCount;
   myErr := FSWrite(gPictFileRefNum, dataLength, dataPtr);
   IF gSpoolPicture <> NIL THEN
   gSpoolPicture^^.picSize := gPictureSize;
END;
```

Gathering Picture Information

You can use the Picture Utilities routines to gather extensive information about pictures and to gather color information about pixel maps. You use the `GetPictInfo` function to gather information about a single picture, and you use the `GetPixMapInfo` function to gather color information about a single pixel map or bitmap. Each of these functions returns color and resolution information in a `PictInfo` record (described on page 7-32). A `PictInfo` record can also contain information about the drawing objects, fonts, and comments in a picture.

You can also survey multiple pictures, pixel maps, and bitmaps for this information. Use the `NewPictInfo` function to begin collecting pictures, pixel maps, and bitmaps for your survey. You also use `NewPictInfo` to specify how you would like the color, comment, and font information for the survey returned to you.

To add the information for a picture to your survey, use the `RecordPictInfo` function. To add the information for a pixel map or a bitmap to your survey, use the `RecordPixMapInfo` function. The `RetrievePictInfo` function collects the information about the pictures, pixel maps, and bitmaps that you have added to your survey. The `RetrievePictInfo` function returns this information in a `PictInfo` record.

For example, to use the ColorSync Utilities to match the colors in a single picture to an output device such as a color printer, an application might find it useful to find the `CMBeginProfile` picture comment, which marks the beginning of a color profile in a `Picture` record. (Color profiles and the ColorSync Utilities are described in *Inside Macintosh: Advanced Color Imaging*.) Listing 7-12 shows an application-defined routine, called `MyGetPICTProfileCount`, that uses `GetPictInfo` to record comments in a `CommentSpec` record (which is described on page 7-30). The `MyGetPICTProfileCount` routine uses the `CommentSpec` record to determine whether any color profiles are included in the picture as picture comments.

Listing 7-12 Looking for color profile comments in a picture

```
FUNCTION MyGetPICTProfileCount (hPICT: PicHandle; VAR count: Integer): OSErr;
VAR
   err:              OSErr;
   thePICTInfo:      PictInfo;
   verb:             Integer;
   colorsRequested:  Integer;
   colorPickMethod:  Integer;
   version:          Integer;
   pCommentSpec:     CommentSpecPtr;
   i:                Integer;
BEGIN
   count := 0;
   verb := recordComments;
   colorsRequested := 0;
   colorPickMethod := systemMethod;
   version := 0;
   err := GetPictInfo(hPICT, thePICTInfo, verb, colorsRequested,
                      colorPickMethod, version);
   IF ((err = noErr) AND (thePICTInfo.commentHandle <> NIL)) THEN
   BEGIN
      pCommentSpec := thePICTInfo.commentHandle^;
      FOR i := 1 TO thePICTInfo.uniqueComments DO
      BEGIN
         IF (pCommentSpec^.ID = CMBeginProfile) THEN
            BEGIN
               count := pCommentSpec^.count;
               LEAVE;
            END;
         pCommentSpec :=
            CommentSpecPtr(ORD4(pCommentSpec)+Sizeof(CommentSpec));
      END;
      {clean up allocations made by GetPictInfo}
      DisposeHandle(Handle(thePICTInfo.commentHandle));
   END;
   MyGetPICTProfileCount := err;
END;
```

If you want information about the colors of a picture or pixel map, you indicate to the Picture Utilities how many colors (up to 256) you want to know about, what method to use for selecting the colors, and whether you want the selected colors returned in a `Palette` record or `ColorTable` record.

The Picture Utilities provide two color-picking methods: one that gives you the most frequently used colors and one that gives you the widest range of colors. Each has advantages in different situations. For example, suppose the picture of a forest image contains 400 colors, of which 300 are greens, 80 are browns, and the rest are a scattering of golden sunlight effects. If you ask for the 250 most used colors, you will probably receive all greens. If you ask for a range of 250 colors, you will receive an assortment stretching from the greens and golds to the browns, including colors in between that might not actually appear in the image. You can also supply a color-picking method of your own, as described in "Application-Defined Routines" beginning on page 7-61.

Your application can then use the color information returned by the Picture Utilities in conjunction with the Palette Manager to provide the best selection of colors for displaying the picture on an 8-bit indexed device.

IMPORTANT
When you ask for color information about a picture, the Picture Utilities take into account only the version 2 and extended version 2 picture opcodes `RGBFgCol`, `RGBBkCol`, `BkPixPat`, `PnPixPat`, `FillPixPat`, and `HiliteColor` (as well as pixel map or bitmap data). Each occurrence of these opcodes is treated as one pixel, regardless of the number and sizes of the objects drawn with that color. If you need an accurate set of colors from a complex picture, create an image of the picture in an offscreen graphics world and call the `GetPixMapInfo` function to obtain color information about that pixel map for that graphics world. ▲

Pictures Reference

This section describes the data structures, routines, and resources provided by QuickDraw for creating and drawing pictures and by the Picture Utilities for gathering information about pictures and pixel maps.

"Data Structures" shows the Pascal data structures for the `Picture`, `OpenCPicParams`, `CommentSpec`, `FontSpec`, and `PictInfo` records.

"QuickDraw and Picture Utilities Routines" describes QuickDraw routines for creating, drawing, and disposing of pictures, and it describes Picture Utilities routines for collecting information about pictures, pixel maps, and bitmaps. "Application-Defined Routines" describes how you can define your own method for selecting colors from pictures and pixel maps.

"Resources" describes the picture (`'PICT'`) resource and the color-picking method (`'cpmt'`) resource.

See Appendix A at the back of this book for a list of picture opcodes.

Data Structures

This section shows the Pascal data structures for the `Picture`, `OpenCPicParams`, `CommentSpec`, `FontSpec`, and `PictInfo` records.

When you use the `OpenCPicture` or `OpenPicture` function, QuickDraw begins collecting your subsequent drawing commands in a `Picture` record. When you use the `GetPicture` function to retrieve a picture stored in a resource, `GetPicture` reads the resource into memory as a `Picture` record.

When you use the `OpenCPicture` function to begin creating a picture, you must pass it information in an `OpenCPicParams` record. This record provides a simple mechanism for specifying resolutions when creating images.

When you use the `GetPictInfo` function, it returns information in a `PictInfo` record. When you gather this information for multiple pictures, pixel maps, or bitmaps, the `RetrievePictInfo` function also returns a `PictInfo` record containing this information.

If you specify the `recordComments` constant in the `verb` parameter to the `GetPictInfo` function or `NewPictInfo` function, your application receives a `PictInfo` record that includes a handle to a `CommentSpec` record. A `CommentSpec` record contains information about the comments in a picture.

If you specify the `recordFontInfo` constant in the `verb` parameter to the `GetPictInfo` function or `NewPictInfo` function, the function returns a `PictInfo` record that includes a handle to a `FontSpec` record. A `FontSpec` record contains information about the fonts in a picture.

Picture

When you use the `OpenCPicture` or `OpenPicture` function (described on page 7-37 and page 7-39, respectively), QuickDraw begins collecting your subsequent drawing commands in a `Picture` record. (You use the `ClosePicture` procedure, described on page 7-42, to complete a picture definition.) When you use the `GetPicture` function (described on page 7-46) to retrieve a picture stored in a resource, `GetPicture` reads the resource into memory as a `Picture` record. (`'PICT'` resources are described on page 7-67.) By using the `DrawPicture` procedure (described on page 7-44), you can draw onscreen the picture defined by the commands stored in the `Picture` record.

A `Picture` record is defined as follows:

```
TYPE Picture =
RECORD
   picSize:    Integer; {for a version 1 picture: its size}
   picFrame:   Rect;    {bounding rectangle for the picture}
   {variable amount of picture data in the form of opcodes}
END;
```

Field descriptions

`picSize` The size of the rest of this record for a version 1 picture. To maintain compatibility with the version 1 picture format, the `picSize` field was not changed for the version 2 picture or extended version 2 formats. The information in this field is useful only for version 1 pictures, which cannot exceed 32 KB in size. Because version 2 and extended version 2 pictures can be much larger than the 32 KB limit imposed by the 2-byte `picSize` field, you should use the Memory Manager function `GetHandleSize` to determine the size of a picture in memory, the File Manager function `PBGetFInfo` to determine the size of a picture in a `'PICT'` file, and the Resource Manager function `MaxSizeResource` to determine the size of a `'PICT'` resource. (See *Inside Macintosh: Memory*, *Inside Macintosh: Files*, and *Inside Macintosh: More Macintosh Toolbox* for more information about these functions.)

`picFrame` The bounding rectangle for the picture defined in the rest of this record. The `DrawPicture` procedure uses this rectangle to scale the picture if you draw it into a destination rectangle of a different size.

Picture comments and compact drawing instructions in the form of picture opcodes compose the rest of this record.

A picture opcode is a number that the `DrawPicture` procedure uses to determine what object to draw or what mode to change for subsequent drawing. For debugging purposes, picture opcodes are listed in Appendix A at the back of this book. Your application generally should not read or write this picture data directly. Instead, your application should use the `OpenCPicture` (or `OpenPicture`), `ClosePicture`, and `DrawPicture` routines to process these opcodes.

The `Picture` record can also contain picture comments. Created by applications using the `PicComment` procedure, picture comments contain data or commands for special processing by output devices, such as PostScript printers. The `PicComment` procedure is described on page 7-40, and picture comments are described in greater detail in Appendix B in this book.

You can use File Manager routines to save the picture in a file of type `'PICT'`, you can use Resource Manager routines to save the picture in a resource of type `'PICT'`, and you can use the Scrap Manager procedure `PutScrap` to store the picture in `'PICT'` scrap format. See the chapter “File Manager” in *Inside Macintosh: Files* and the chapters “Resource Manager” and “Scrap Manager” in *Inside Macintosh: More Macintosh Toolbox* for more information about saving files, resources, and scrap data.

OpenCPicParams

When you use the `OpenCPicture` function (described on page 7-37) to begin creating a picture, you must pass it information in an `OpenCPicParams` record. This record provides a simple mechanism for specifying resolutions when creating images. For example, applications that create pictures from scanned images can specify resolutions higher than 72 dpi for these pictures in `OpenCPicParams` records.

An `OpenCPicParams` record is defined as follows:

```
TYPE OpenCPicParams =
RECORD
   srcRect:    Rect;       {optimal bounding rectangle for }
                           { displaying picture at resolution }
                           { indicated in hRes, vRes fields}
   hRes:       Fixed;      {best horizontal resolution; }
                           { $00480000 specifies 72 dpi}
   vRes:       Fixed;      {best vertical resolution; }
                           { $00480000 specifies 72 dpi}
   version:    Integer;    {set to -2}
   reserved1:  Integer;    {reserved; set to 0}
   reserved2:  LongInt;    {reserved; set to 0}
END;
```

Field descriptions

`srcRect` The optimal bounding rectangle for the resolution indicated by the next two fields. When you later call the `DrawPicture` procedure (described on page 7-44) to play back the saved picture, you supply a destination rectangle, and `DrawPicture` scales the picture so that it is completely aligned with the destination rectangle. To display a picture at a resolution other than that specified in the next two fields, your application should compute an appropriate destination rectangle by scaling its width and height by the following factor:

scale factor = destination resolution / source resolution

`hRes` The best horizontal resolution for the picture. Notice that this value is of type `Fixed`—a value of $0048000 specifies a horizontal resolution of 72 dpi.

`vRes` The best vertical resolution for the picture. Notice that this value is of type `Fixed`—a value of $00480000 specifies a horizontal resolution of 72 dpi.

`version` Always set this field to –2.

`reserved1` Reserved; set to 0.

`reserved2` Reserved; set to 0.

CommentSpec

If you specify the `recordComments` constant in the `verb` parameter to the `GetPictInfo` function (described on page 7-47) or the `NewPictInfo` function (described on page 7-53), you receive a `PictInfo` record (described beginning on page 7-32) that includes in its `commentHandle` field a handle to an array of `CommentSpec` records. The `uniqueComments` field of the `PictInfo` record indicates the number of `CommentSpec` records in this array.

The `CommentSpec` record is defined as follows:

```
TYPE CommentSpec =   {comment specification record}
RECORD
   count:   Integer; {number of times this type of comment }
                     { occurs in the picture or survey}
   ID:      Integer; {value identifying this type of comment}
END;
```

Field descriptions

`count` The number of times this kind of picture comment occurs in the picture specified to the `GetPictInfo` function or in all the pictures examined with the `NewPictInfo` function.

`ID` The value set in the `kind` parameter when the picture comment was created with the `PicComment` procedure. The `PicComment` procedure is described on page 7-40. The values of many common IDs are listed in Appendix B in this book.

When you are finished using the information returned in a `CommentSpec` record, you should use the `DisposeHandle` procedure (described in *Inside Macintosh: Memory*) to dispose of the memory allocated to it.

Listing 7-12 on page 7-25 illustrates how to count the number of picture comments by examining a `CommentSpec` record.

FontSpec

If you specify the `recordFontInfo` constant in the `verb` parameter to the `GetPictInfo` function (described on page 7-47) or the `NewPictInfo` function (described on page 7-53), your application receives a `PictInfo` record (described beginning on page 7-32) that includes in its `fontHandle` field a handle to an array of `FontSpec` records. The `uniqueFonts` field of the `PictInfo` record indicates the number of `FontSpec` records in this array. (For bitmap fonts, a font is a complete set of glyphs in one size, typeface, and style—for example, 12-point Geneva italic. For outline fonts, a font is a complete set of glyphs in one typeface and style—for example, Geneva italic.)

The `FontSpec` record is defined as follows:

```
TYPE FontSpec =         {font specification record}
RECORD
   pictFontID: Integer; {font ID as stored in the picture}
   sysFontID:  Integer; {font family ID}
   size:       ARRAY[0..3] OF LongInt;
                        {each bit set from 1 to 127 indicates a }
                        { point size at that value; if bit 0 is }
                        { set, then a size larger than 127 }
                        { points is found}
   style:      Integer; {styles used for this font family}
   nameOffset: LongInt; {offset to font name stored in the }
                        { data structure indicated by the }
                        { fontNamesHandle field of the PictInfo }
                        { record}
END;
```

Field descriptions

`pictFontID` The ID number of the font as it is stored in the picture.

`sysFontID` The number that identifies the resource file (of type `'FOND'`) that specifies the font family. Every font family—which consists of a complete set of fonts for one typeface including all available styles and sizes of the glyphs in that typeface—has a unique font family ID, in a range of values that determines the script system to which the font family belongs.

`size` The point sizes of the fonts in the picture. The field contains 128 bits, in which a bit is set for each point size encountered, from 1 to 127 points. Bit 0 is set if a size larger than 127 is found.

`style` The styles for this font family at any of its sizes. The values in this field can also be represented with the `Style` data type:

```
TYPE
   StyleItem = (bold, italic, underline, outline,
                shadow, condense, extend);
   Style = SET OF StyleItem;
```

`nameOffset` The offset into the list of font names (indicated by the `fontNamesHandle` field of the `PictInfo` record) at which the name for this font family is stored. A font name, such as Geneva, is given to a font family to distinguish it from other font families.

When you are finished using the information returned in a `FontSpec` record, you should use the `DisposeHandle` procedure (described in *Inside Macintosh: Memory*) to dispose of the memory allocated to it.

See the chapter "Font Manager" in *Inside Macintosh: Text* for more information about fonts.

PictInfo

When you use the `GetPictInfo` function (described on page 7-47) to collect information about a picture, or when you use the `GetPixMapInfo` function (described on page 7-50) to collect color information about a pixel map or bitmap, the function returns the information in a `PictInfo` record. When you gather this information for multiple pictures, pixel maps, or bitmaps, the `RetrievePictInfo` function (described on page 7-58) also returns a `PictInfo` record containing this information.

Initially, all of the fields in a new `PictInfo` record are set to `NIL`. Relevant fields are set to appropriate values depending on the information you request using the Picture Utilities functions as described in this chapter.

The `PictInfo` record is defined as follows:

```
TYPE PictInfo =
RECORD
   version:          Integer;          {Picture Utilities version number}
   uniqueColors:     LongInt;          {total colors in survey}
   thePalette:       PaletteHandle;    {handle to a Palette record--NIL for }
                                       { a bitmap in a basic graphics port}
   theColorTable:    CTabHandle;       {handle to a ColorTable record--NIL }
                                       { for a bitmap in a basic graphics }
                                       { port}
   hRes:             Fixed;            {best horizontal resolution (dpi)}
   vRes:             Fixed;            {best vertical resolution (dpi)}
   depth:            Integer;          {greatest pixel depth}
   sourceRect:       Rect;             {optimal bounding rectangle for }
                                       { picture for display at resolution }
                                       { specified in hRes and vRes fields}
   textCount:        LongInt;          {number of text strings in picture(s)}
   lineCount:        LongInt;          {number of lines in picture(s)}
   rectCount:        LongInt;          {number of rectangles in picture(s)}
   rRectCount:       LongInt;          {number of rounded rectangles in }
                                       { picture(s)}
   ovalCount:        LongInt;          {number of ovals in picture(s)}
   arcCount:         LongInt;          {number of arcs and wedges in }
                                       { picture(s)}
```

```
   polyCount:        LongInt;          {number of polygons in picture(s)}
   regionCount:      LongInt;          {number of regions in picture(s)}
   bitMapCount:      LongInt;          {number of bitmaps}
   pixMapCount:      LongInt;          {number of pixel maps}
   commentCount:     LongInt;          {number of comments in picture(s)}
   uniqueComments:   LongInt;          {number of different comments }
                                       { (by ID) in picture(s)}
   commentHandle:    CommentSpecHandle;{handle to an array of CommentSpec }
                                       { records for picture(s)}
   uniqueFonts:      LongInt;          {number of fonts in picture(s)}
   fontHandle:       FontSpecHandle;   {handle to an array of FontSpec }
                                       { records for picture(s)}
   fontNamesHandle:  Handle;           {handle to list of font names for }
                                       { picture(s)}
   reserved1:        LongInt;
   reserved2:        LongInt;
END;
```

Field descriptions

`version` The version number of the Picture Utilities, currently set to 0.

`uniqueColors` The number of colors in the picture specified to the `GetPictInfo` function, or the number of colors in the pixel map or bitmap specified to the `GetPixMapInfo` function, or the total number of colors for all the pictures, pixel maps, and bitmaps returned by the `RetrievePictInfo` function. The number of colors returned in this field is limited by the accuracy of the Picture Utilities' color bank for color storage. See "Application-Defined Routines" beginning on page 7-61 for information about the Picture Utility's color bank and about how you can create your own for selecting colors.

`thePalette` A handle to the resulting `Palette` record if you specified to the `GetPictInfo`, `GetPixMapInfo`, or `NewPictInfo` function that colors be returned in a `Palette` record. That `Palette` record contains either the number of colors you specified to the function or—if there aren't that many colors in the pictures, pixel maps, or bitmaps—the number of colors found. Depending on the constant you pass in the `verb` parameter to the function, the `Palette` record contains either the most used or the widest range of colors in the pictures, pixel maps, and bitmaps. On Macintosh computers running basic QuickDraw only, this field is always returned as `NIL`. See the chapter "Palette Manager" in *Inside Macintosh: Advanced Color Imaging* for more information about `Palette` records.

`theColorTable`	A handle to the resulting `ColorTable` record if you specified to the `GetPictInfo`, `GetPixMapInfo`, or `NewPictInfo` function that colors be returned in a `ColorTable` record. If the pictures, pixel maps, or bitmaps contain fewer colors found than you specified to the function, the unused entries in the `ColorTable` record are filled with black. Depending on the constant you pass in the `verb` parameter to the function, the `ColorTable` record contains either the most used or the widest range of colors in the pictures, pixel maps, and bitmaps. The chapter "Color QuickDraw" in this book describes `ColorTable` records. On Macintosh computers running basic QuickDraw only, this field is always returned as `NIL`. If a picture has more than 256 colors or has pixel depths of 32 bits, then Color QuickDraw translates the colors in the `ColorTable` record to 16-bit depths. In such a case, the returned colors might have a slight loss of resolution, and the `uniqueColors` field reflects the number of colors distinguishable at that pixel depth.
`hRes`	The horizontal resolution of the current picture, pixel map, or bitmap retrieved by the `GetPictInfo` or `GetPixMapInfo` function; the greatest horizontal resolution from all pictures, pixel maps, and bitmaps retrieved by the `RetrievePictInfo` function.
`vRes`	The vertical resolution of the current picture, pixel map, or bitmap retrieved by the `GetPictInfo` or `GetPixMapInfo` function; the greatest vertical resolution of all pictures, pixel maps, and bitmaps retrieved by the `RetrievePictInfo` function. Note that although the values of the `hRes` and `vRes` fields are usually the same, they don't have to be.
`depth`	The pixel depth of the picture specified to the `GetPictInfo` function or the pixel map specified to the `GetPixMapInfo` function. When you use the `RetrievePictInfo` function, this field contains the deepest pixel depth of all pictures or pixel maps retrieved by the function.
`sourceRect`	The optimal bounding rectangle for displaying the picture at the resolution indicated by the `hRes` and `vRes` fields. The upper-left corner of the rectangle is always (0,0). Pictures created with the `OpenCPicture` function have the `hRes`, `vRes`, and `sourceRect` fields built into their `Picture` records. For pictures created by `OpenPicture`, the `hRes` and `vRes` fields are set to 72 dpi, and the source rectangle is calculated using the `picFrame` field of the `Picture` record for the picture.
`textCount`	The number of text strings in the picture specified to the `GetPictInfo` function, or the total number of text objects in all the pictures retrieved by the `RetrievePictInfo` function. For pixel maps and bitmaps specified to `GetPixMapInfo` or `RetrievePictInfo`, this field is set to 0.
`lineCount`	The number of lines in the picture specified to the `GetPictInfo` function, or the total number of lines in all the pictures retrieved by the `RetrievePictInfo` function. For pixel maps and bitmaps, this field is set to 0.

`rectCount`	The number of rectangles in the picture specified to the `GetPictInfo` function, or the total number of rectangles in all the pictures retrieved by the `RetrievePictInfo` function. For pixel maps and bitmaps, this field is set to 0.
`rRectCount`	The number of rounded rectangles in the picture specified to the `GetPictInfo` function, or the total number of rounded rectangles in all the pictures retrieved by the `RetrievePictInfo` function. For pixel maps and bitmaps, this field is set to 0.
`ovalCount`	The number of ovals in the picture specified to the `GetPictInfo` function, or the total number of ovals in all the pictures retrieved by the `RetrievePictInfo` function. For pixel maps and bitmaps, this field is set to 0.
`arcCount`	The number of arcs and wedges in the picture specified to the `GetPictInfo` function, or the total number of arcs and wedges in all the pictures retrieved by the `RetrievePictInfo` function. For pixel maps and bitmaps, this field is set to 0.
`polyCount`	The number of polygons in the picture specified to the `GetPictInfo` function, or the total number of polygons in all the pictures retrieved by the `RetrievePictInfo` function. For pixel maps and bitmaps, this field is set to 0.
`regionCount`	The number of regions in the picture specified to the `GetPictInfo` function, or the total number of regions in all the pictures retrieved by the `RetrievePictInfo` function. For pixel maps and bitmaps, this field is set to 0.
`bitMapCount`	The total number of bitmaps in the survey.
`pixMapCount`	The total number of pixel maps in the survey.
`commentCount`	The number of comments in the picture specified to the `GetPictInfo` function, or the total number of comments in all the pictures retrieved by the `RetrievePictInfo` function. This field is valid only if you specified to the `GetPictInfo` or `NewPictInfo` function that comments be returned in a `CommentSpec` record, described on page 7-30. For pixel maps and bitmaps, this field is set to 0.
`uniqueComments`	The number of picture comments that have different IDs in the picture specified to the `GetPictInfo` function, or the total number of picture comments with different IDs in all the pictures retrieved by the `RetrievePictInfo` function. (The values for many common IDs are listed in Appendix B, “Using Picture Comments for Printing,” in this book.) This field is valid only if you specify that comments be returned in a `CommentSpec` record. For pixel maps and bitmaps, this field is set to 0.
`commentHandle`	A handle to an array of `CommentSpec` records, described on page 7-30. For pixel maps and bitmaps, this field is set to `NIL`.

`uniqueFonts` The number of different fonts in the picture specified to the `GetPictInfo` function, or the total number of different fonts in all the pictures retrieved by the `RetrievePictInfo` function. For bitmap fonts, a font is a complete set of glyphs in one size, typeface, and style—for example, 12-point Geneva italic. For outline fonts, a font is a complete set of glyphs in one typeface and style—for example, Geneva italic.

This field is valid only if you specify that fonts be returned in a `FontSpec` record, which is described on page 7-30. For pixel maps and bitmaps, this field is set to 0.

`fontHandle` A handle to a list of `FontSpec` records, described on page 7-30. For pixel maps and bitmaps, this field is set to `NIL`.

`fontNamesHandle`
A handle to the names of the fonts in the picture retrieved by the `GetPictInfo` function or the pictures retrieved by the `RetrievePictInfo` function. The offset to a particular name is stored in the `nameOffset` field of the `FontSpec` record for that font. A font name is a name, such as Geneva, given to one font family to distinguish it from other font families.

When you are finished with this information, be sure to dispose of it. You can dispose of `Palette` records by using the `DisposePalette` procedure (which is described in the chapter "Palette Manager" in *Inside Macintosh: Advanced Color Imaging*). You can dispose of `ColorTable` records by using the `DisposeCTable` procedure (described in the chapter "Color QuickDraw" in this book). You can dispose of other allocations with the `DisposeHandle` procedure (described in *Inside Macintosh: Memory*).

QuickDraw and Picture Utilities Routines

This section describes QuickDraw routines for creating, drawing, and disposing of pictures, and it describes Picture Utilities routines for collecting information about pictures, pixel maps, and bitmaps.

Creating and Disposing of Pictures

Use the `OpenCPicture` function to begin defining a picture; `OpenCPicture` collects your subsequent drawing commands—which are described in the other chapters of this book—in a `Picture` record. You can use the `PicComment` procedure to include picture comments in your picture definition. To complete the collection of drawing and picture comment commands that define your picture, use the `ClosePicture` procedure. When you are finished using a picture not stored in a `'PICT'` resource, use the `KillPicture` procedure to release its memory. (To release the memory for a picture stored in a `'PICT'` resource, use the Resource Manager procedure `ReleaseResource`.)

The `OpenCPicture` function works on all Macintosh computers running System 7. Pictures created with the `OpenCPicture` function can be drawn on all versions of Macintosh system software. The `OpenPicture` function, which was created for earlier versions of system software, is described here for completeness.

OpenCPicture

To begin defining a picture in extended version 2 format, use the `OpenCPicture` function.

```
FUNCTION OpenCPicture (newHeader: OpenCPicParams): PicHandle;
```

`newHeader` An `OpenCPicParams` record, which is defined as follows (see page 7-29 for a description of the `OpenCPicParams` data type):

```
TYPE OpenCPicParams =
RECORD
   srcRect:    Rect;    {optimal bounding rectangle }
                        { for displaying picture at }
                        { resolution indicated in }
                        { hRes, vRes fields}
   hRes:       Fixed;   {best horizontal resolution; }
                        { $00480000 specifies 72 dpi}
   vRes:       Fixed;   {best vertical resolution; }
                        { $00480000 specifies 72 dpi}
   version:    Integer; {set to -2}
   reserved1:  Integer; {reserved; set to 0}
   reserved2:  LongInt; {reserved; set to 0}
END;
```

DESCRIPTION

The `OpenCPicture` function returns a handle to a new `Picture` record (described on page 7-27). Use the `OpenCPicture` function to begin defining a picture; `OpenCPicture` collects your subsequent drawing commands in this record. When defining a picture, you can use all other QuickDraw drawing routines described in this book, with the exception of `CopyMask`, `CopyDeepMask`, `SeedFill`, `SeedCFill`, `CalcMask`, and `CalcCMask`. (Nor can you use the `PlotCIcon` procedure, described in *Inside Macintosh: More Macintosh Toolbox*.)

You can also use the `PicComment` procedure (described on page 7-40) to include picture comments in your picture definition.

The `OpenCPicture` function creates pictures in the extended version 2 format. This format permits your application to specify resolutions when creating images.

Use the `OpenCPicParams` record you pass in the `newHeader` parameter to specify the horizontal and vertical resolution for the picture, and specify an optimal bounding rectangle for displaying the picture at this resolution. When you later call the `DrawPicture` procedure (described on page 7-44) to play back the saved picture, you supply a destination rectangle, and `DrawPicture` scales the picture so that it is completely aligned with the destination rectangle. To display a picture at a resolution other than that at which it was created, your application should compute an appropriate destination rectangle by scaling its width and height by the following factor:

scale factor = destination resolution / source resolution

For example, if a picture was created at 300 dpi and you want to display it at 75 dpi, then your application should compute the destination rectangle width and height as 1/4 of those of the picture's bounding rectangle.

The `OpenCPicture` function calls the `HidePen` procedure, so no drawing occurs on the screen while the picture is open (unless you call the `ShowPen` procedure just after `OpenCPicture`, or you called `ShowPen` previously without balancing it by a call to `HidePen`).

Use the handle returned by `OpenCPicture` when referring to the picture in subsequent routines, such as the `DrawPicture` procedure.

After defining the picture, close it by using the `ClosePicture` procedure, described on page 7-42. To draw the picture, use the `DrawPicture` procedure, described on page 7-44.

After creating the picture, your application can use the `GetPictInfo` function (described on page 7-47) to gather information about it. The `PictInfo` record (described on page 7-32) returned by `GetPictInfo` returns the picture's resolution and optimal bounding rectangle.

SPECIAL CONSIDERATIONS

When creating a picture, you should generally use the `ClosePicture` procedure to finish it before you open the Printing Manager with the `PrOpen` procedure. There are two main reasons for this. First, you should allow the printing driver to use as much memory as possible. Second, the Printing Manager creates its own type of graphics port—one that replaces the standard QuickDraw drawing operations stored in the `grafProcs` field of a `CGrafPort` or `GrafPort` record; to avoid unexpected results when creating a picture, you should draw into a graphics port created with QuickDraw instead of drawing into a printing port created by the Printing Manager.

After calling `OpenCPicture`, be sure to finish your picture definition by calling `ClosePicture` before you call `OpenCPicture` again. You cannot nest calls to `OpenCPicture`.

Always use the `ClipRect` procedure to specify a clipping region appropriate for your picture before you call `OpenCPicture`. If you do not use `ClipRect` to specify a clipping region, `OpenCPicture` uses the clipping region specified in the current graphics port. If the clipping region is very large (as it is when a graphics port is initialized) and you scale the picture when drawing it, the clipping region can become invalid when `DrawPicture` scales the clipping region—in which case, your picture will

not be drawn. On the other hand, if the graphics port specifies a small clipping region, part of your drawing may be clipped when you draw it. Setting a clipping region equal to the port rectangle of the current graphics port, as shown in Listing 7-1 on page 7-11, always sets a valid clipping region.

The `OpenCPicture` function may move or purge memory.

SEE ALSO

The `PrOpen` procedure is described in the chapter "Printing Manager" in this book. The `ClipRect` procedure is described in the chapter "Basic QuickDraw" in this book. The `ShowPen` and `HidePen` procedures are described in the chapter "QuickDraw Drawing" in this book.

Listing 7-1 on page 7-11 illustrates the use of the `OpenCPicture` function.

OpenPicture

The `OpenPicture` function, which was created for earlier versions of system software, is described here for completeness. To create a picture, you should use the `OpenCPicture` function, which allows you to specify resolutions for your pictures, as explained in the previous routine description.

```
FUNCTION OpenPicture (picFrame: Rect): PicHandle;
```

`picFrame` The bounding rectangle for the picture. The `DrawPicture` procedure uses this rectangle to scale the picture if you draw it into a destination rectangle of a different size.

DESCRIPTION

The `OpenPicture` function returns a handle to a new `Picture` record (described on page 7-27). You can use the `OpenPicture` function to begin defining a picture; `OpenPicture` collects your subsequent drawing commands in this record. When defining a picture, you can use all other QuickDraw drawing routines described in this book, with the exception of `CopyMask`, `CopyDeepMask`, `SeedFill`, `SeedCFill`, `CalcMask`, and `CalcCMask`. (Nor can you use the `PlotCIcon` procedure, described in *Inside Macintosh: More Macintosh Toolbox*.) You can also use the `PicComment` procedure (described on page 7-40) to include picture comments in your picture definition.

The `OpenPicture` function creates pictures in the version 2 format on computers with Color QuickDraw when the current graphics port is a color graphics port. Pictures created in this format support color drawing operations at 72 dpi. On computers supporting only basic QuickDraw, or when the current graphics port is a basic graphics port, this function creates pictures in version 1 format. Pictures created in version 1 format support only black-and-white drawing operations at 72 dpi.

Use the handle returned by `OpenPicture` when referring to the picture in subsequent routines, such as the `DrawPicture` procedure.

The `OpenPicture` function calls the `HidePen` procedure, so no drawing occurs on the screen while the picture is open (unless you call the `ShowPen` procedure just after `OpenPicture` or you called `ShowPen` previously without balancing it by a call to `HidePen`).

After defining the picture, close it by using the `ClosePicture` procedure, described on page 7-42. To draw the picture, use the `DrawPicture` procedure, described on page 7-44.

SPECIAL CONSIDERATIONS

The version 2 and version 1 picture formats support only 72-dpi resolution. The `OpenCPicture` function creates pictures in the extended version 2 format. The extended version 2 format, which is created by the `OpenCPicture` function on all Macintosh computers running System 7, permits your application to specify additional resolutions when creating images.

See the description of the `OpenCPicture` function for its list of special considerations, all of which apply to `OpenPicture`.

Version 1 pictures are limited to 32 KB. You can determine the picture size while it's being formed by calling the Memory Manager function `GetHandleSize`.

PicComment

To insert a picture comment into a picture that you are defining or into your printing code, use the `PicComment` procedure.

```
PROCEDURE PicComment (kind,dataSize: Integer;
                      dataHandle: Handle);
```

`kind` The type of comment. Because the vast majority of picture comments are interpreted by printer drivers, the constants that you can supply in this parameter—and values they represent—are listed in Appendix B, "Using Picture Comments for Printing," in this book.

`dataSize` Size of any additional data passed in the `dataHandle` parameter. Data sizes for the various kinds of picture comments are listed in Appendix B, "Using Picture Comments for Printing," in this book. If no additional data is used, specify 0 in this parameter.

`dataHandle`
A handle to additional data, if used. If no additional data is used, specify `NIL` in this parameter.

DESCRIPTION

When used after your application begins creating a picture with the `OpenCPicture` (or `OpenPicture`) function, the `PicComment` procedure inserts the specified comment into the `Picture` record. When sent to a printer driver after your application uses the `PrOpenPage` procedure, `PicComment` passes the data or commands in the specified comment directly to the printer.

Picture comments contain data or commands for special processing by output devices, such as printers. For example, using the `SetLineWidth` comment, your application can draw hairlines—which are not available with standard QuickDraw calls—on any PostScript LaserWriter printer and on the QuickDraw LaserWriter SC printer.

Usually printer drivers process picture comments, but applications can also do so. For your application to process picture comments, it must replace the `StdComment` procedure pointed to by the `commentProc` field of the `CQDProcs` or `QDProcs` record, which in turn is pointed to by the `grafProcs` field of a `CGrafPort` or `GrafPort` record. The default `StdComment` procedure provided by QuickDraw does no comment processing whatsoever. You can use the `SetStdCProcs` procedure to assist you in changing the `CQDProcs` record, and you can use the `SetStdProcs` procedure to assist you in changing the `QDProcs` record.

If you create and process your own picture comments, you should define comments so that they contain information that identifies your application (to avoid using the same comments as those used by Apple or by other third-party products). You should define a comment as an `ApplicationComment` comment type with a `kind` value of 100. The first 4 bytes of the data for the comment should specify your application's signature. (See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for information about application signatures.) You can use the next 2 bytes to identify the type of comment—that is, to specify a `kind` value to your own application.

Suppose your application signature were `'WAVE'`, and you wanted to use the value 128 to identify a `kind` value to your own application. You would supply values to the `kind` and `data` parameters to `PicComment` as follows:

`kind` = 100; `data` = `'WAVE'` [4 bytes] + 128 [2 bytes] + additional data [*n* bytes]

Your application can then parse the first 6 bytes of the comment to determine whether and how to process the rest of the data in the comment. It is up to you to publish information about your comments if you wish them to be understood and used by other applications.

SPECIAL CONSIDERATIONS

These former picture comments are now obsolete: `SetGrayLevel`, `ResourcePS`, `PostScriptFile`, and `TextIsPostScript`.

The `PicComment` procedure may move or purge memory.

SEE ALSO

See Appendix B, "Using Picture Comments for Printing," in this book for information about using picture comments to print with features that are unavailable with QuickDraw. See the chapter "QuickDraw Drawing" in this book for information about the `QDProcs` record and the `StdComment` and `SetStdProcs` procedures. See the chapter "Color QuickDraw" in this book for information about the `CQDProcs` record and the `SetStdCProcs` procedure.

ClosePicture

To complete the collection of drawing commands and picture comments that define your picture, use the `ClosePicture` procedure.

```
PROCEDURE ClosePicture;
```

DESCRIPTION

The `ClosePicture` procedure stops collecting drawing commands and picture comments for the currently open picture. You should perform one and only one call to `ClosePicture` for every call to the `OpenCPicture` (or `OpenPicture`) function.

The `ClosePicture` procedure calls the `ShowPen` procedure, balancing the call made by `OpenCPicture` (or `OpenPicture`) to the `HidePen` procedure.

SEE ALSO

The `ShowPen` and `HidePen` procedures are described in the chapter "QuickDraw Drawing" in this book.

Listing 7-1 on page 7-11 illustrates the use of the `ClosePicture` procedure.

KillPicture

To release the memory occupied by a picture not stored in a `'PICT'` resource, use the `KillPicture` procedure.

```
PROCEDURE KillPicture (myPicture: PicHandle);
```

`myPicture` A handle to the picture whose memory can be released.

DESCRIPTION

The `KillPicture` procedure releases the memory occupied by the picture whose handle you pass in the `myPicture` parameter. Use this only when you're completely finished with a picture.

SPECIAL CONSIDERATIONS

If you use the Window Manager procedure `SetWindowPic` to store a picture handle in the window record, you can use the Window Manager procedure `DisposeWindow` or `CloseWindow` to release the memory allocated to the picture; these procedures automatically call `KillPicture` for the picture.

If the picture is stored in a `'PICT'` resource, you must use the Resource Manager procedure `ReleaseResource` instead of `KillPicture`. The Window Manager procedures `DisposeWindow` and `CloseWindow` will not delete it; instead, you must call `ReleaseResource` before calling `DisposeWindow` or `CloseWindow`.

The `KillPicture` procedure may move or purge memory.

SEE ALSO

The `ReleaseResource` procedure is described in the chapter "Resource Manager" in *Inside Macintosh: Macintosh Toolbox*, and the `SetWindowPic`, `DisposeWindow`, and `CloseWindow` procedures are described in the chapter "Window Manager," in *Inside Macintosh: Macintosh Toolbox Essentials*.

Drawing Pictures

To draw a picture, use the `DrawPicture` procedure. You must access a picture through its handle. When creating pictures, the `OpenCPicture` and `OpenPicture` functions (described beginning on page 7-37) return their handles. You can use the `GetPicture` function to get a handle to a QuickDraw picture stored in a `'PICT'` resource.

Note
To get a handle to a QuickDraw picture stored in a `'PICT'` file, you must use File Manager routines, as described in "Drawing a Picture Stored in a 'PICT' File" beginning on page 7-13. To get a picture stored in the scrap, use the Scrap Manager procedure `GetScrap` to get a handle to its data and then coerce this handle to one of type `PicHandle`, as shown in Listing 7-6 on page 7-17. ◆

DrawPicture

To draw a picture on any type of output device, use the `DrawPicture` procedure.

```
PROCEDURE DrawPicture (myPicture: PicHandle; dstRect: Rect);
```

`myPicture` A handle to the picture to be drawn.

`dstRect` A destination rectangle, specified in coordinates local to the current graphics port, in which to draw the picture. The `DrawPicture` procedure shrinks or expands the picture as necessary to align the borders of its bounding rectangle with the rectangle you specify in this parameter. To display a picture at a resolution other than that at which it was created, your application should compute an appropriate destination rectangle by scaling its width and height by the following factor:

scale factor = destination resolution / source resolution

For example, if a picture was created at 300 dpi and you want to display it at 75 dpi, then your application should compute the destination rectangle width and height as 1/4 of those of the picture's bounding rectangle. Your application can use the `GetPictInfo` function (described on page 7-47) to gather information about a picture. The `PictInfo` record (described on page 7-32) returned by `GetPictInfo` returns the picture's resolution in its `hRes` and `vRes` fields. The `sourceRect` field contains the bounding rectangle for displaying the image at its optimal resolution.

DESCRIPTION

Within the rectangle that you specify in the `dstRect` parameter, the `DrawPicture` procedure draws the picture that you specify in the `myPicture` parameter.

The `DrawPicture` procedure passes any picture comments to the `StdComment` procedure pointed to by the `commentProc` field of the `CQDProcs` or `QDProcs` record, which in turn is pointed to by the `grafProcs` field of a `CGrafPort` or `GrafPort` record. The default `StdComment` procedure provided by QuickDraw does no comment processing whatsoever. If you want to process picture comments when drawing a picture, you can use the `SetStdCProcs` procedure to assist you in changing the `CQDProcs` record, and you can use the `SetStdProcs` procedure to assist you in changing the `QDProcs` record.

SPECIAL CONSIDERATIONS

Always use the `ClipRect` procedure to specify a clipping region appropriate for your picture before defining it with the `OpenCPicture` (or `OpenPicture`) function. If you do not use `ClipRect` to specify a clipping region, `OpenCPicture` uses the clipping region specified in the current graphics port. If the clipping region is very large (as it is when a graphics port is initialized) and you want to scale the picture, the clipping region

can become invalid when `DrawPicture` scales the clipping region—in which case, your picture will not be drawn. On the other hand, if the graphics port specifies a small clipping region, part of your drawing may be clipped when `DrawPicture` draws it. Setting a clipping region equal to the port rectangle of the current graphics port, as shown in Listing 7-1 on page 7-11, always sets a valid clipping region.

When it scales, `DrawPicture` changes the size of the font instead of scaling the bits. However, the widths used by bitmap fonts are not always linear. For example, the 12-point width isn't exactly 1/2 of the 24-point width. This can cause lines of text to become slightly longer or shorter as the picture is scaled. The difference is often insignificant, but if you are trying to draw a line of text that fits exactly into a box (a spreadsheet cell, for example), the difference can become noticeable to the user—most typically, at print time. The easiest way to avoid such problems is to specify a destination rectangle that is the same size as the bounding rectangle for the picture. Otherwise, your application may need to directly process the opcodes in the picture instead of using `DrawPicture`.

You may also have disappointing results if the fonts contained in an image are not available on the user's system. Before displaying a picture, your application may want to use the Picture Utilities to determine what fonts are contained in the picture, and then use Font Manager routines to determine whether the fonts are available on the user's system. If they are not, you can use Dialog Manager routines to display an alert box warning the user of display problems.

If there is insufficient memory to draw a picture in Color QuickDraw, the `QDError` function (described in the chapter "Color QuickDraw" in this book) returns the result code `noMemForPictPlaybackErr`.

The `DrawPicture` procedure may move or purge memory.

SEE ALSO

Listing 7-1 on page 7-11 illustrates how to use `DrawPicture` after creating a picture while your application is running; Listing 7-2 on page 7-13 illustrates how to use `DrawPicture` after reading in a picture stored in a `'PICT'` file; Listing 7-6 on page 7-17 illustrates how to use `DrawPicture` after reading in a picture stored in the scrap; and Listing 7-8 on page 7-20 illustrates how to use `DrawPicture` after reading in a picture stored in a `'PICT'` resource.

See the chapter "Font Manager" in *Inside Macintosh: Text* for information about Font Manager routines; see the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information about Dialog Manager routines.

GetPicture

Use the `GetPicture` function to get a handle to a picture stored in a `'PICT'` resource.

```
FUNCTION GetPicture (picID: Integer): PicHandle;
```

`picID` The resource ID for a `'PICT'` resource.

DESCRIPTION

The `GetPicture` function returns a handle to the picture stored in the `'PICT'` resource with the ID that you specify in the `picID` parameter. You can pass this handle to the `DrawPicture` procedure (described on page 7-44) to draw the picture stored in the resource.

The `GetPicture` function calls the Resource Manager procedure `GetResource` as follows:

```
GetResource('PICT',picID)
```

If the resource can't be read, `GetPicture` returns `NIL`.

SPECIAL CONSIDERATIONS

To release the memory occupied by a picture stored in a `'PICT'` resource, use the Resource Manager procedure `ReleaseResource`.

The `GetPicture` function may move or purge memory.

SEE ALSO

The `GetResource` and `ReleaseResource` procedures are described in the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*. The `'PICT'` resource is described on page 7-67.

Collecting Picture Information

You can use the Picture Utilities routines described in this section to gather extensive information about pictures and to gather color information about pixel maps and bitmaps. On an 8-bit indexed device, for example, you might want your application to determine the 256 most-used colors in a picture composed of millions of colors. Your application can then use the Palette Manager (as described in *Inside Macintosh: Advanced Color Imaging*) to make these colors available for the window in which your application needs to draw the picture.

You use the `GetPictInfo` function to gather information about a single picture, and you use the `GetPixMapInfo` function to gather color information about a single pixel map or bitmap. Each of these functions returns color and resolution information in a `PictInfo` record. A `PictInfo` record for a picture also contains additional information, such as the resolution of the picture, and information about the fonts and comments contained in the picture.

You can also survey multiple pictures, pixel maps, and bitmaps for this information. Use the `NewPictInfo` function to begin collecting pictures, pixel maps, and bitmaps for your survey. You also use `NewPictInfo` to specify how you would like the color, comment, and font information for the survey returned to you.

To add the information for a picture to your survey, use the `RecordPictInfo` function. To add the information for a pixel map or a bitmap to your survey, use the `RecordPixMapInfo` function. The `RetrievePictInfo` function collects the information about the pictures, pixel maps, and bitmaps that you have added to the survey. The `RetrievePictInfo` function returns this information in a `PictInfo` record.

When you are finished with this information, use the `DisposePictInfo` function to dispose of the private data structures allocated by the `NewPictInfo` function.

Note
The Picture Utilities also collect information from black-and-white pictures and bitmaps, and they are supported in System 7 even by computers running only basic QuickDraw. However, when collecting color information on a computer running only basic QuickDraw, the Picture Utilities return `NIL` instead of a handle to a `Palette` or `ColorTable` record. ◆

GetPictInfo

Use the `GetPictInfo` function to gather information about a single picture.

```
FUNCTION GetPictInfo (thePictHandle: PicHandle;
                      VAR thePictInfo: PictInfo; verb: Integer;
                      colorsRequested: Integer;
                      colorPickMethod: Integer;
                      version: Integer): OSErr;
```

`thePictHandle`
A handle to a picture.

`thePictInfo`
A pointer to a `PictInfo` record, which will hold information about the picture. The `PictInfo` record is described on page 7-32.

verb A value indicating what type of information you want GetPictInfo to return in the PictInfo record. You can use any or all of the following constants or the sum of the integers they represent:

```
CONST
returnColorTable  = 1;  {return a ColorTable record}
returnPalette     = 2;  {return a Palette record}
recordComments    = 4;  {return comment information}
recordFontInfo    = 8;  {return font information}
suppressBlackAndWhite
                  = 16; {don't include black and }
                        { white with returned colors}
```

Because the Palette Manager adds black and white when creating a Palette record, you can specify the number of colors you want minus 2 in the colorsRequested parameter and specify the suppressBlackAndWhite constant in the verb parameter when gathering colors destined for a Palette record or a screen.

colorsRequested
From 1 to 256, the number of colors you want in the ColorTable or Palette record returned via the PictInfo record. If you are not requesting colors (that is, if you pass the recordComments or recordFontInfo constant in the verb parameter), this function does not return colors, in which case you may instead pass 0 here.

colorPickMethod
The method by which colors are selected for the ColorTable or Palette record returned via the PictInfo record. You can use one of the following constants or the integer it represents:

```
CONST
systemMethod   = 0;  {let Picture Utilities choose }
                     { the method (currently they }
                     { always choose popularMethod)}
popularMethod  = 1;  {return most frequently used }
                     { colors}
medianMethod   = 2;  {return a weighted distribution }
                     { of colors}
```

You can also create your own color-picking method in a resource file of type 'cpmt' and pass its resource ID in the colorPickMethod parameter. The resource ID must be greater than 127.

version Always set this parameter to 0.

DESCRIPTION

In the `PictInfo` record to which the parameter `thePictInfo` points, the `GetPictInfo` function returns information about the picture you specify in the `thePictHandle` parameter. Initially, all of the fields in a new `PictInfo` record are set to `NIL`. Relevant fields are set to appropriate values depending on the information you request using the `GetPictInfo` function.

Use the `verb` parameter to specify whether you want color information (in a `ColorTable` record, a `Palette` record, or both), whether you want picture comment information, and whether you want font information. If you want color information, be sure to use the `colorPickMethod` parameter to specify the method by which to select colors.

The Picture Utilities provide two color-picking methods: one (specified by the `popularMethod` constant) that gives you the most frequently used colors and one (specified by the `medianMethod` constant) that gives you the widest range of colors. Each has advantages in different situations. For example, suppose the picture of a forest image contains 400 colors, of which 300 are greens, 80 are browns, and the rest are a scattering of golden sunlight effects. If you ask for the 250 most used colors, you will probably receive all greens. If you ask for a range of 250 colors, you will receive an assortment stretching from the greens and golds to the browns, including colors in between that might not actually appear in the image. If you specify the `systemMethod` constant, the Picture Utilities choose the method; currently they always choose `popularMethod`. You can also supply a color-picking method of your own.

If your application uses more than one color-picking method, it should present the user with a choice of which method to use.

When you are finished with the information in the `PictInfo` record, be sure to dispose of it. Use the Memory Manager procedure `DisposeHandle` to dispose of the `PictInfo`, `CommentSpec`, and `FontSpec` records. Dispose of the `Palette` record by using the `DisposePalette` procedure. Dispose of the `ColorTable` record by using the `DisposeCTable` procedure.

SPECIAL CONSIDERATIONS

When you ask for color information, `GetPictInfo` takes into account only the version 2 and extended version 2 picture opcodes `RGBFgCol`, `RGBBkCol`, `BkPixPat`, `PnPixPat`, `FillPixPat`, and `HiliteColor` (as well as pixel map or bitmap data). Each occurrence of these opcodes is treated as 1 pixel, regardless of the number and sizes of the objects drawn with that color. If you need an accurate set of colors from a complex picture, create an image of the picture in an offscreen pixel map, and then call the `GetPixMapInfo` function (described on page 7-50) to obtain color information about that pixel map.

The `GetPictInfo` function returns a bit depth of 1 on QuickTime-compressed `'PICT'` files. However, when QuickTime is installed, QuickTime decompresses and displays the image correctly.

The `GetPictInfo` function may move or purge memory.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetPictInfo` function are

Trap macro	Selector
`_Pack15`	$0800

RESULT CODES

`pictInfoVersionErr`	–11000	Version number not 0
`pictInfoVerbErr`	–11002	Invalid verb combination specified
`cantLoadPickMethodErr`	–11003	Custom pick method not in resource chain
`colorsRequestedErr`	–11004	Number out of range or greater than that passed to `NewPictInfo`
`pictureDataErr`	–11005	Invalid picture data

SEE ALSO

The `PictInfo` record is described on page 7-32, the `CommentSpec` record is described on page 7-30, and the `FontSpec` record is described on page 7-30. The `ColorTable` record is described in the chapter "Color QuickDraw" in this book; the `Palette` record is described in the chapter "Palette Manager" in *Inside Macintosh: Advanced Color Imaging*. See "Application-Defined Routines" beginning on page 7-61 for more information about creating your own color-picking method for the `colorPickMethod` parameter.

The `DisposePalette` procedure is described in the chapter "Palette Manager" in *Inside Macintosh: Advanced Color Imaging*. The `DisposeCTable` procedure is described in the chapter "Color QuickDraw" in this book. The `DisposeHandle` procedure is described in the chapter "Memory Manager" in *Inside Macintosh: Memory.*

Listing 7-12 on page 7-25 illustrates the use of the `GetPictInfo` function.

GetPixMapInfo

Use the `GetPixMapInfo` function to gather color information about a single pixel map or bitmap.

```
FUNCTION GetPixMapInfo (thePixMapHandle: PixMapHandle;
                        VAR thePictInfo: PictInfo; verb: Integer;
                        colorsRequested: Integer;
                        colorPickMethod: Integer;
                        version: Integer): OSErr;
```

`thePixMapHandle`
A handle to a pixel map or bitmap.

thePictInfo
: A pointer to a PictInfo record, which will hold information about a pixel map or bitmap. The PictInfo record is described on page 7-32.

verb
: A value indicating whether you want color information returned in a ColorTable record, a Palette record, or both. You can also request that black and white not be included among the returned colors. You can use any or all of the following constants or the sum of the integers they represent:

```
CONST
returnColorTable  = 1;  {return a ColorTable record}
returnPalette     = 2;  {return a Palette record}
suppressBlackAndWhite
                  = 16; {don't include black and }
                        { white with returned colors}
```

Because the Palette Manager adds black and white when creating a Palette record, you can specify the number of colors you want minus 2 in the colorsRequested parameter and specify the constant suppressBlackAndWhite in the verb parameter when gathering colors destined for a Palette record or a screen.

colorsRequested
: From 1 to 256, the number of colors you want in the ColorTable or Palette record returned via the PictInfo record.

colorPickMethod
: The method by which colors are selected for the ColorTable or Palette record returned via the PictInfo record. You can use one of the following constants or the integer it represents:

```
CONST
systemMethod   = 0;  {let Picture Utilities choose }
                     { the method (currently they }
                     { always choose popularMethod)}
popularMethod  = 1;  {return most frequently used }
                     { colors}
medianMethod   = 2;  {return a weighted distribution }
                     { of colors}
```

You can also create your own color-picking method in a resource file of type 'cpmt' and pass its resource ID in the colorPickMethod parameter. The resource ID must be greater than 127.

version
: Always set this parameter to 0.

DESCRIPTION

For the pixel map (or bitmap) whose handle you pass in the `thePixMapHandle` parameter, the `GetPixMapInfo` function returns color information in the `PictInfo` record that you point to in the parameter `thePictInfo`. Initially, all of the fields in a new `PictInfo` record are set to `NIL`. Relevant fields are set to appropriate values depending on the information you request using the `GetPixMapInfo` function.

Use the `verb` parameter to specify whether you want color information returned in a `ColorTable` record, a `Palette` record, or both, and use the `colorPickMethod` parameter to specify the method by which to select colors.

The Picture Utilities provide two color-picking methods: one (specified by the `popularMethod` constant) that gives you the most frequently used colors and one (specified by the `medianMethod` constant) that gives you the widest range of colors. If you specify the `systemMethod` constant, the Picture Utilities choose the method; currently they always choose `popularMethod`. You can also supply a color-picking method of your own.

When you are finished with the information in the `PictInfo` record, be sure to dispose of it. Use the Memory Manager procedure `DisposeHandle` to dispose of the `PictInfo` record. Dispose of the `Palette` record by using the `DisposePalette` procedure. Dispose of the `ColorTable` record by using the `DisposeCTable` procedure.

SPECIAL CONSIDERATIONS

The `GetPixMapInfo` function may move or purge memory.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetPixMapInfo` function are

Trap macro	Selector
`_Pack15`	$0801

RESULT CODES

`pictInfoVersionErr`	–11000	Version number not 0
`pictInfoVerbErr`	–11002	Invalid verb combination specified
`cantLoadPickMethodErr`	–11003	Custom pick method not in resource chain
`colorsRequestedErr`	–11004	Number out of range or greater than that passed to `NewPictInfo`

SEE ALSO

See "Application-Defined Routines" beginning on page 7-61 for more information about creating your own color-picking method for the `colorPickMethod` parameter. The `PictInfo` record is described on page 7-32; the `PixMapHandle` data type and the `ColorTable` record are described in the chapter "Color QuickDraw" in this book; the `Palette` record is described in *Inside Macintosh: Advanced Color Imaging*.

The `DisposePalette` procedure is described in *Inside Macintosh: Advanced Color Imaging*. The `DisposeCTable` procedure is described in the chapter "Color QuickDraw" in this book. The `DisposeHandle` procedure is described in the chapter "Memory Manager" in *Inside Macintosh: Memory.*

NewPictInfo

You can survey multiple pictures for such information as colors, picture comments, and fonts, and you can survey multiple pixel maps and bitmaps for color information. Use the `NewPictInfo` function to begin collecting pictures, pixel maps, and bitmaps for your survey.

```
FUNCTION NewPictInfo (VAR thePictInfoID: PictInfoID;
                      verb: Integer; colorsRequested: Integer;
                      colorPickMethod: Integer;
                      version: Integer): OSErr;
```

`thePictInfoID`
A unique value that denotes your collection of pictures, pixel maps, or bitmaps.

`verb` A value indicating what type of information you want the `RetrievePictInfo` function (described on page 7-58) to return in a `PictInfo` record (described on page 7-32). When collecting information about pictures, you can use any or all of the following constants or the sum of the integers they represent:

```
CONST
returnColorTable  = 1;  {return a ColorTable record}
returnPalette     = 2;  {return a Palette record}
recordComments    = 4;  {return comment information}
recordFontInfo    = 8;  {return font information}
suppressBlackAndWhite
                  = 16; {don't include black and }
                        { white with returned colors}
```

The constants `recordComments` and `recordFontInfo` and the values they represent have no effect when gathering information about the pixel maps and bitmaps included in your survey.

Because the Palette Manager adds black and white when creating a palette, you can specify the number of colors you want minus 2 in the `colorsRequested` parameter and specify the constant `suppressBlackAndWhite` in the `verb` parameter when gathering colors destined for a `Palette` record or a screen.

colorsRequested
: From 1 to 256, the number of colors you want included in the `ColorTable` or `Palette` record returned by the `RetrievePictInfo` function via a `PictInfo` record.

colorPickMethod
: The method by which colors are selected for the `ColorTable` or `Palette` record included in the `PictInfo` record returned by the `RetrievePictInfo` function. You can use one of the following constants or the integer it represents:

```
CONST
systemMethod   = 0;  {let Picture Utilities choose }
                     { the method (currently they }
                     { always choose popularMethod)}
popularMethod  = 1;  {return most frequently used }
                     { colors}
medianMethod   = 2;  {return a weighted distribution }
                     { of colors}
```

: You can also create your own color-picking method in a resource file of type `'cpmt'` and pass its resource ID in the `colorPickMethod` parameter. The resource ID must be greater than 127.

version
: Always set this parameter to 0.

DESCRIPTION

In the `thePictInfoID` parameter, the `NewPictInfo` function returns a unique ID number for use when surveying multiple pictures, pixel maps, and bitmaps for information.

To add the information for a picture to your survey, use the `RecordPictInfo` function, which is described next. To add the information for a pixel map or a bitmap to your survey, use the `RecordPixMapInfo` function, which is described on page 7-57. For each of these functions, you identify the survey with the ID number returned by `NewPictInfo`.

Use the `RetrievePictInfo` function (described on page 7-58) to return information about the pictures, pixel maps, and bitmaps in the survey. Again, you identify the survey with the ID number returned by `NewPictInfo`. The `RetrievePictInfo` function returns your requested information in a `PictInfo` record.

Use the `verb` parameter for `NewPictInfo` to specify whether you want to gather comment or font information for the pictures in the survey. If you want to gather color information, use the `verb` parameter for `NewPictInfo` to specify whether you want this information in a `ColorTable` record, a `Palette` record, or both. The `PictInfo` record returned by the `RetrievePictInfo` function will then include a handle to a `ColorTable` record or a `Palette` record, or handles to both. If you want color information, be sure to use the `colorPickMethod` parameter to specify the method by which to select colors.

The Picture Utilities provide two color-picking methods: one (specified by the `popularMethod` constant) that gives you the most frequently used colors and one (specified by the `medianMethod` constant) that gives you the widest range of colors. If you specify the `systemMethod` constant, the Picture Utilities choose the method; currently they always choose `popularMethod`. You can also supply a color-picking method of your own.

SPECIAL CONSIDERATIONS

The `NewPictInfo` function may move or purge memory.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `NewPictInfo` function are

Trap macro	Selector
`_Pack15`	$0602

RESULT CODES

`pictInfoVersionErr`	–11000	Version number not 0
`pictInfoVerbErr`	–11002	Invalid verb combination specified
`cantLoadPickMethodErr`	–11003	Custom pick method not in resource chain
`colorsRequestedErr`	–11004	Number out of range or greater than that passed to `NewPictInfo`

SEE ALSO

The `PictInfo` record is described on page 7-32, the `CommentSpec` record is described on page 7-30, and the `FontSpec` record is described on page 7-30. The `ColorTable` record is described in the chapter "Color QuickDraw" in this book; the `Palette` record is described in *Inside Macintosh: Advanced Color Imaging*. See "Application-Defined Routines" beginning on page 7-61 for more information about creating your own color-picking method for the `colorPickMethod` parameter.

RecordPictInfo

To add a picture to an informational survey of multiple pictures, use the `RecordPictInfo` function.

```
FUNCTION RecordPictInfo (thePictInfoID: PictInfoID;
                         thePictHandle: PicHandle): OSErr;
```

`thePictInfoID`
The ID number—returned by the `NewPictInfo` function—that identifies the survey to which you are adding the picture. The `NewPictInfo` function is described on page 7-53.

`thePictHandle`
A handle to the picture being added to the survey.

DESCRIPTION

The `RecordPictInfo` function adds the picture you specify in the parameter `thePictHandle` to the survey of pictures identified by the parameter `thePictInfoID`. Use `RecordPictInfo` repeatedly to add additional pictures to your survey.

After you have collected all of the pictures you need, use the `RetrievePictInfo` function, described on page 7-58, to return information about pictures in the survey.

SPECIAL CONSIDERATIONS

When you ask for color information, `RecordPictInfo` takes into account only the version 2 and extended version picture opcodes `RGBFgCol`, `RGBBkCol`, `BkPixPat`, `PnPixPat`, `FillPixPat`, and `HiliteColor`. Each occurrence of these opcodes is treated as 1 pixel, regardless of the number and sizes of the objects drawn with that color. If you need an accurate set of colors from a complex picture, create an image of the picture in an offscreen pixel map, and then call the `GetPixMapInfo` function (described on page 7-50) to obtain color information about that pixel map.

The `RecordPictInfo` function may move or purge memory.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `RecordPictInfo` function are

Trap macro	Selector
`_Pack15`	$0403

RESULT CODES

`pictInfoIDErr`	–11001	Invalid picture information ID
`pictureDataErr`	–11005	Invalid picture data

RecordPixMapInfo

To add a pixel map or bitmap to an informational survey of multiple pixel maps and bitmaps, use the `RecordPictInfo` function.

```
FUNCTION RecordPixMapInfo (thePictInfoID: PictInfoID;
                           thePixMapHandle: PixMapHandle): OSErr;
```

`thePictInfoID`
The ID number—returned by the `NewPictInfo` function—that identifies the survey to which you are adding the pixel map or bitmap. The `NewPictInfo` function is described on page 7-53.

`thePixMapHandle`
A handle to a pixel map (or bitmap) to be added to the survey.

DESCRIPTION

The `RecordPixMapInfo` function adds the pixel map or bitmap you specify in the parameter `thePixMapHandle` to the survey identified by the parameter `thePictInfoID`. Use `RecordPictInfo` repeatedly to add additional pixel maps and bitmaps to your survey.

After you have collected all of the images you need, use the `RetrievePictInfo` function, described on page 7-58, to return information about all the images in the survey.

SPECIAL CONSIDERATIONS

The `RecordPixMapInfo` function may move or purge memory.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `RecordPixMapInfo` function are

Trap macro	Selector
`_Pack15`	$0404

RESULT CODES

`pictInfoIDErr`	–11001	Invalid picture information ID
`pictureDataErr`	–11005	Invalid picture data

RetrievePictInfo

Use the `RetrievePictInfo` function to return information about all the pictures, pixel maps, and bitmaps included in a survey.

```
FUNCTION RetrievePictInfo (thePictInfoID: PictInfoID;
                           VAR thePictInfo: PictInfo;
                           colorsRequested: Integer): OSErr;
```

`thePictInfoID`
The ID number—returned by the `NewPictInfo` function—that identifies the survey of pictures, pixel maps, and bitmaps. The `NewPictInfo` function is described on page 7-53.

`thePictInfo`
A pointer to the `PictInfo` record that holds information about the pictures or images in the survey. The `PictInfo` record is described on page 7-32.

`colorsRequested`
From 1 to 256, the number of colors you want returned in the `ColorTable` or `Palette` record included in the `PictInfo` record.

DESCRIPTION

In a `PictInfo` record that you point to in the parameter `thePictInfo`, the `RetrievePictInfo` function returns information about all of the pictures and images collected in the survey that you specify in the parameter `thePictInfoID`.

After using the `NewPictInfo` function to create a new survey, and then using `RecordPictInfo` to add pictures to your survey and `RecordPixMapInfo` to add pixel maps and bitmaps to your survey, you can call `RetrievePictInfo`.

When you are finished with the information in the `PictInfo` record, be sure to dispose of it. You can dispose of the `Palette` record by using the `DisposePalette` procedure. You can dispose of the `ColorTable` record by using the `DisposeCTable` procedure. You can dispose of other allocations with the `DisposeHandle` procedure. You should also use the `DisposePictInfo` function (described next) to dispose of the private data structures created by the `NewPictInfo` function.

SPECIAL CONSIDERATIONS

The `RetrievePictInfo` function may move or purge memory.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `RetrievePictInfo` function are

Trap macro	Selector
`_Pack15`	$0505

RESULT CODES

`pictInfoIDErr`	–11001	Invalid picture information ID
`colorsRequestedErr`	–11004	Number out of range or greater than that passed to `NewPictInfo`

SEE ALSO

The `DisposePalette` procedure is described in *Inside Macintosh: Advanced Color Imaging*. The `DisposeCTable` procedure is described in the chapter "Color QuickDraw" in this book. The `DisposeHandle` procedure is described in the chapter "Memory Manager" in *Inside Macintosh: Memory.*

DisposePictInfo

When you are finished gathering information from a survey of pictures, pixel maps, or bitmaps, use the `DisposePictInfo` function to dispose of the private data structures allocated by the `NewPictInfo` function. The `DisposePictInfo` function is also available as the `DisposPictInfo` function.

```
FUNCTION DisposePictInfo (thePictInfoID: PictInfoID): OSErr;
```

`thePictInfoID`
The unique identifier returned by `NewPictInfo`.

DESCRIPTION

The `DisposePictInfo` function disposes of the private data structures allocated by the `NewPictInfo` function, which is described on page 7-53.

The `DisposePictInfo` function does not dispose of any of the handles returned to you in a `PictInfo` record by the `RetrievePictInfo` function, which is described on page 7-58. Instead, you can dispose of a `Palette` record by using the `DisposePalette` procedure. You can dispose of a `ColorTable` record by using the `DisposeCTable` procedure. You can dispose of other allocations with the `DisposeHandle` procedure.

SPECIAL CONSIDERATIONS

The `DisposePictInfo` function may move or purge memory.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DisposePictInfo` function are

Trap macro	Selector
`_Pack15`	$0206

RESULT CODE

`pictInfoIDErr`	–11001	Invalid picture information ID

SEE ALSO

The `DisposePalette` procedure is described in *Inside Macintosh: Advanced Color Imaging*. The `DisposeCTable` procedure is described in the chapter "Color QuickDraw" in this book. The `DisposeHandle` procedure is described in the chapter "Memory Manager" in *Inside Macintosh: Memory.*

Application-Defined Routines

As described in "Collecting Picture Information" beginning on page 7-46, your application can use the `GetPictInfo`, `GetPixMapInfo`, and `NewPictInfo` functions to gather information about pictures, pixel maps, and bitmaps. Each of these functions can gather up to 256 colors in `ColorTable` and `Palette` records. In the `colorPickMethod` parameter to these functions, you specify how they should select which colors to gather. These Picture Utilities functions provide two color-picking methods: the first selects the most frequently used colors, and the second selects a weighted distribution of the existing colors.

You can also create your own color-picking method. You must compile it as a resource of type `'cpmt'` and write its entry point in assembly language. To use this color-picking method (`'cpmt'`) resource, pass its resource ID (which must be greater than 127) in the `colorPickMethod` parameter to these Picture Utilities functions. These functions call your color-picking method's entry point and pass one of four possible selectors in register D0. These functions pass their parameters on the stack. As shown in Table 7-1, each selector requires your color-picking method to call a different routine, which should return its results in register D0.

Table 7-1 Routine selectors for an application-defined color-picking method

D0 value	Routine to call
0	`MyInitPickMethod`
1	`MyRecordColors`
2	`MyCalcColorTable`
3	`MyDisposeColorPickMethod`

Your color-picking method (`'cpmt'`) resource should include a routine that specifies its **color bank** (that is, the structure into which all the colors of a picture, pixel map, or bitmap are gathered) and allocates whatever data your color-picking method needs. This routine is explained next as a Pascal function declared as `MyInitPickMethod`.

Your `MyInitPickMethod` function can let the Picture Utilities generate a color bank consisting of a **histogram** (that is, frequency counts of each color) to a resolution of 5 bits per color. Or, your `MyInitPickMethod` function can specify that your application has its own custom color bank—for example, a histogram to a resolution of 8 bits per color.

If you create your own custom color bank, your `'cpmt'` resource should include a routine that gathers and stores colors; this routine is described on page 7-64 as a Pascal function declared as `MyRecordColors`.

For the number of colors your application requests using the Picture Utilities, your `'cpmt'` resource should include a routine that determines which colors to select from the color bank and then fills an array of `ColorSpec` records with those colors; this routine is described on page 7-65 as a Pascal function declared as `MyCalcColorTable`. The Picture Utilities function that your application initially called then returns these

colors in a `Palette` record or `ColorTable` record, as specified by your application when it first called the Picture Utilities function.

Your `'cpmt'` resource should include a routine that releases the memory allocated by your `MyInitPickMethod` routine. The routine that releases memory is described on page 7-67 as a Pascal function declared as `MyDisposeColorPickMethod`.

If your routines return an error, that error is passed back to the `GetPictInfo`, `GetPixMapInfo`, or `NewPictInfo` function, which in turn passes the error to your application as a function result.

MyInitPickMethod

Your color-picking method (`'cpmt'`) resource should include a routine that specifies its color bank and allocates whatever data your color-picking method needs. Here is how you would declare this routine if it were a Pascal function named `MyInitPickMethod`:

```
FUNCTION MyInitPickMethod (colorsRequested: Integer;
                           VAR dataRef: LongInt;
                           VAR colorBankType: Integer): OSErr;
```

`colorsRequested`
The number of colors requested by your application to be gathered for examination in a `ColorTable` or `Palette` record.

`dataRef` A handle to any data needed by your color-picking method; that is, if your application allocates and uses additional data, it should return a handle to it in this parameter.

`colorBankType`
The type of color bank your color-picking method uses. Your `MyInitPickMethod` routine should return one of three valid color bank types, which it can represent with one of these constants:

```
CONST
colorBankIsCustom       = -1; {gathers colors into a }
                              { custom color bank}
colorBankIsExactAnd555  = 0;  {gathers exact colors }
                              { if there are less }
                              { than 256 unique }
                              { colors in picture; }
                              { otherwise gathers }
                              { colors for picture }
                              { in a 5-5-5 histogram}
colorBankIs555          = 1;  {gathers colors into a }
                              { 5-5-5 histogram}
```

Return the `colorBankIs555` constant in this parameter if you want to let the Picture Utilities gather the colors for a picture or a pixel map into a 5-5-5 histogram. When you return the `colorBankIs555` constant, the Picture Utilities call your `MyCalcColorTable` routine with a pointer to the color bank (that is, to the 5-5-5 histogram). Your `MyCalcColorTable` routine (described on page 7-65) selects whatever colors it needs from this color bank. Then the Picture Utilities function called by your application returns these colors in a `Palette` record, a `ColorTable` record, or both, as requested by your application.

Return the `ColorBankIsExactAnd555` constant in this parameter to make the Picture Utilities return exact colors if there are less than 256 unique colors in the picture; otherwise, the Picture Utilities gather the colors for the picture in a 5-5-5 histogram, just as they do when you return the `colorBankIs555` constant. If the picture or pixel map has fewer colors than your application requests when it calls a Picture Utilities function, the Picture Utilities function returns all of the colors contained in the color bank. If the picture or pixel map contains more colors than your application requests, the Picture Utilities call your `MyCalcColorTable` routine to select which colors to return.

Return the `colorBankIsCustom` constant in this parameter if you want to implement your own color bank for storing the colors in a picture or a pixel map. For example, because the 5-5-5 histogram that the Picture Utilities provide gathers colors to a resolution of 5 bits per color, your application may want to create a histogram with a resolution of 8 bits per color. When you return the `colorBankIsCustom` constant, the Picture Utilities call your `MyRecordColors` routine (explained in the next routine description) to create this color bank. The Picture Utilities also call your `MyCalcColorTable` routine to select colors from this color bank.

DESCRIPTION

Your `MyInitPickMethod` routine should allocate whatever data your color-picking method needs and store a handle to your data in the location pointed to by the `dataRef` parameter. In the `colorBankType` parameter, your `MyInitPickMethod` routine must also return the type of color bank your color-picking method uses for color storage. If your `MyInitPickMethod` routine generates any error, it should return the error as its function result.

The 5-5-5 histogram that the Picture Utilities provide if you return the `ColorBankIs555` or `ColorBankIsExactAnd555` constant in the `colorBankType` parameter is like a reversed `cSpecArray` record, which is an array of `ColorSpec` records. (The `cSpecArray` and `ColorSpec` records are described in the chapter "Color QuickDraw" in this book.) This 5-5-5 histogram is an array of 32,768 integers, where the *index* into the array is the color: 5 bits of red, followed by 5 bits of green, followed by 5 bits of blue. Each *entry* in the array is the number of colors in the picture that are approximated by the index color for that entry.

For example, suppose there were three instances of the following color in the pixel map:

Red = %1101 1010 1010 1110

Green = %0111 1010 1011 0001

Blue = %0101 1011 0110 1010

This color would be represented by index % 0 11011-01111-01011 (in hexadecimal, $6DEB), and the value in the histogram at this index would be 3, because there are three instances of this color.

MyRecordColors

When you return the `colorBankIsCustom` constant in the `colorBankType` parameter to your `MyInitPickMethod` function (described in the preceding section), your color-picking method (`'cpmt'`) resource must include a routine that creates this color bank; for example, your application may want to create a histogram with a resolution of 8 bits per color. Here is how you would declare this routine if it were a Pascal function named `MyRecordColors`:

```
FUNCTION MyRecordColors (dataRef: LongInt;
                         colorsArray: RGBColorArray;
                         colorCount: LongInt;
                         VAR uniqueColors: LongInt): OSErr;
```

`dataRef` A handle to any data your method needs. Your application initially creates this handle using the `MyInitPickMethod` routine (explained in the preceding section).

`colorsArray`
An array of `RGBColor` records. (`RGBColor` records are described in the chapter "Color QuickDraw" in this book.) Your `MyRecordColors` routine should store the color information for this array of `RGBColor` records in a data structure of type `RGBColorArray`. You should define the `RGBColorArray` data type as follows:

```
TYPE RGBColorArray  =  ARRAY[0..0] OF RGBColor;
```

`colorCount`
The number of colors in the array specified in the `colorsArray` parameter.

`uniqueColors`
Upon input: the number of unique colors already added to the array in the `colorsArray` parameter. (The Picture Utilities functions call your `MyRecordColors` routine once for every color in the picture, pixel map, or bitmap.) Your `MyRecordColors` routine must calculate the number of unique colors (to the resolution of the color bank) that are added by this call. Your `MyRecordColors` routine should add this amount to the value passed upon input in this parameter and then return the sum in this parameter.

DESCRIPTION

Your `MyRecordColors` routine should store each color encountered in a picture or pixel into its own color bank. The Picture Utilities call `MyRecordColors` only if your `MyInitPickMethod` routine returns the constant `colorBankIsCustom` in the `colorBankType` parameter. The Picture Utilities functions call `MyRecordColors` for all the colors in the picture, pixel map, or bitmap. If your `MyRecordColors` routine generates any error, it should return the error as its function result.

MyCalcColorTable

Your color-picking method (`'cpmt'`) resource should include a routine that selects as many colors as are requested by your application from the color bank for a picture or pixel map and then fills these colors into an array of `ColorSpec` records.

Here is how you would declare this routine if it were a Pascal function named `MyCalcColorTable`:

```
FUNCTION MyCalcColorTable (dataRef: LongInt;
                           colorsRequested: Integer;
                           colorBankPtr: Ptr;
                           VAR resultPtr: CSpecArray): OSErr;
```

`dataRef` A handle to any data your method needs. Your application initially creates this handle using the `MyInitPickMethod` routine (explained on page 7-62).

`colorsRequested`
The number of colors requested by your application to be gathered for examination in a `ColorTable` or `Palette` record.

`colorBankPtr`
If your `MyInitPickMethod` routine (described on page 7-62) returned either the `colorBankIsExactAnd555` or `colorBankIs555` constant, then this parameter contains a pointer to the 5-5-5 histogram that describes all of the colors in the picture, pixel map, or bitmap being examined. (The format of the 5-5-5 histogram is explained in the routine description for the `MyInitPickMethod` routine.) Your `MyCalcColorTable` routine should examine these colors and then, using its own criterion for selecting the colors, fill in an array of `ColorSpec` records with the number of colors specified in the `colorsRequested` parameter.

If your `MyInitPickMethod` routine returned the `colorBankIsCustom` constant, then the value passed in this parameter is invalid. In this case, your `MyCalcColorTable` routine should use the custom color bank that your application created (as explained in the routine description for the `MyRecordColors` routine on page 7-64) for filling in an array `of ColorSpec` records with the number of colors specified in the `colorsRequested` parameter.

Your `MyCalcColorTable` function should return a pointer to this array of `ColorSpec` records in the next parameter.

`resultPtr` A pointer to the array of `ColorSpec` records to be filled with the number of colors specified in the `colorsRequested` parameter. The Picture Utilities function that your application initially called places these colors in a `Palette` record or `ColorTable` record, as specified by your application.

DESCRIPTION

Selecting from the color bank created for the picture, bitmap, or pixel map being examined, your `MyCalcColorTable` routine should fill an array of `ColorSpec` records with the number of colors requested in the `colorsRequested` parameter and return this array in the `resultPtr` parameter. If your `MyCalcColorTable` routine generates any error, it should return the error as its function result.

If more colors are requested than the picture contains, fill the remaining entries with black (0000 0000 0000).

The `colorBankPtr` parameter is of type `Ptr` because the data stored in the color bank is of the type specified by your `MyInitPickMethod` routine (described on page 7-62). Thus, if you specified `colorBankIs555` in the `colorBankType` parameter, the color bank would be an array of integers. However, if the Picture Utilities support other data types in the future, the `colorBankPtr` parameter could point to completely different data types.

SPECIAL CONSIDERATIONS

Always coerce the value passed in the `colorBankPtr` parameter to a pointer to an integer. In the future you may need to coerce this value to a pointer of the type you specify in your `MyInitPickMethod` function.

MyDisposeColorPickMethod

Your `'cpmt'` resource should include a routine that releases the memory allocated by your `MyInitPickMethod` routine (which is described on page 7-62). Here is how you would declare this routine if it were a Pascal function named `MyDisposeColorPickMethod`:

```
FUNCTION MyDisposeColorPickMethod (dataRef: LongInt): OSErr;
```

`dataRef` A handle to any data your method needs. Your application initially creates this handle using the `MyInitPickMethod` routine.

DESCRIPTION

Your `MyDisposeColorPickMethod` routine should release any memory that you allocated in your `MyInitPickMethod` routine. If your `MyDisposeColorPickMethod` routine generates any error, it should return the error as its function result.

Resources

This section describes the picture (`'PICT'`) resource and the color-picking method (`'cpmt'`) resource. You can use the `'PICT'` resource to save pictures in the resource fork of your application or document files. You can assemble your own color-picking method for use by the Picture Utilities in a `'cpmt'` resource.

The Picture Resource

A picture (`'PICT'`) resource contains QuickDraw drawing instructions that can be played back using the `DrawPicture` procedure.

You may find it useful to store pictures in the resource fork of your application or document file. For example, when the user chooses the About command in the Apple menu for your application, you might wish to display a window containing your company's logo. Or, if yours is a page-layout application, you might want to store all the images created by the user for a document as resources in the document file.

You can use high-level tools like the ResEdit resource editor, available from APDA, to create and store images as `'PICT'` resources for distribution with your files.

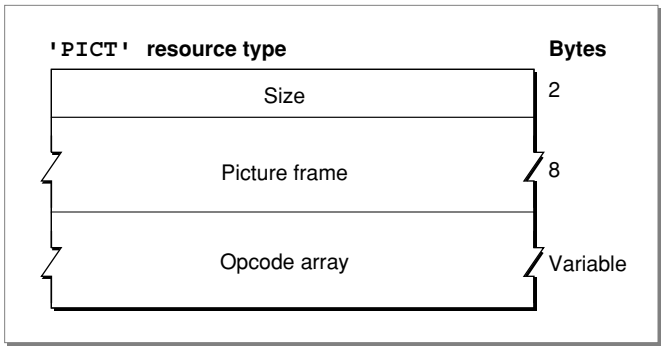
To save a picture in a `'PICT'` resource while your application is running, you should use Resource Manager routines, such as `FSpOpenResFile` (to open your application's resource fork), `ChangedResource` (to change an existing `'PICT'` resource), `AddResource` (to add a new `'PICT'` resource), `WriteResource` (to write the data to the resource), and `CloseResFile` and `ReleaseResource` (to conclude saving the resource). These routines are described in the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*.

All 'PICT' resources must be marked purgeable, and they must have resource IDs greater than 127.

If you examine the compiled version of a 'PICT' resource, as represented in Figure 7-4, you find that it contains the following elements:

- The size of the resource—if the resource contains a picture created in the version 1 format. Because version 2 and extended version 2 pictures can be much larger than the 32 KB limit imposed by the size of this element, you should use the Resource Manager function MaxSizeResource (described in *Inside Macintosh: More Macintosh Toolbox*) to determine the size of a picture in version 2 and extended version 2 format.
- The bounding rectangle for the picture. The DrawPicture procedure uses this rectangle to scale the picture when you draw it into a destination rectangle of a different size.
- An array of picture opcodes. A picture opcode is a number that the DrawPicture procedure uses to determine what object to draw or what mode to change for subsequent drawing. For debugging purposes, picture opcodes are listed in Appendix A at the back of this book. Your application generally should not read or write this picture data directly. Instead, you should use the OpenCPicture (or OpenPicture), ClosePicture, and DrawPicture routines to process these opcodes.

Figure 7-4 Structure of a compiled picture ('PICT') resource



To retrieve a 'PICT' resource, specify its resource ID to the GetPicture function, described on page 7-46, which returns a handle to the picture. Listing 7-8 on page 7-20 illustrates an application-defined routine that retrieves and draws a picture stored as a resource.

Appendix A, "Picture Opcodes," shows examples of disassembled picture resources.

The Color-Picking Method Resource

The resource type for an assembled color-picking routine is 'cpmt'. It must have a resource ID greater than 127. The resource data is the assembled code of the routine. See "Application-Defined Routines" beginning on page 7-61 for information about creating a color-picking method resource.

Summary of Pictures and the Picture Utilities

Pascal Summary

Constants

```
CONST
   {color-picking methods}
   systemMethod   = 0;  {let Picture Utilities choose the method (currently }
                        { they always choose popularMethod)}
   popularMethod  = 1;  {return most frequently used colors}
   medianMethod   = 2;  {return a weighted distribution of colors}

   {picture information to be returned by Picture Utilities}
   returnColorTable        = 1;  {return a ColorTable record}
   returnPalette           = 2;  {return a Palette record}
   recordComments          = 4;  {return comment information}
   recordFontInfo          = 8;  {return font information}
   suppressBlackAndWhite   = 16; {don't include black and }
                                 { white with returned colors}

   {color bank types}
   colorBankIsCustom       = -1; {gathers colors into a custom color bank}
   colorBankIsExactAnd555  = 0;  {gathers exact colors if there are less }
                                 { than 256 unique colors in picture; }
                                 { otherwise gathers colors for picture }
                                 { in a 5-5-5 histogram}
   colorBankIs555          = 1;  {gathers colors in a 5-5-5 histogram}
```

Data Types

```
TYPE
   PicPtr      = ^Picture;
   PicHandle   = ^PicPtr;
   Picture     =
   RECORD                  {picture record}
      picSize:    Integer; {for a version 1 picture: its size}
```

```
   picFrame:   Rect;    {bounding rectangle for the picture}
   {variable amount of picture data in the form of opcodes}
END;

OpenCPicParams =
RECORD
   srcRect:    Rect;       {optimal bounding rectangle for displaying }
                           { picture at resolution indicated in hRes, }
                           { vRes fields}
   hRes:       Fixed;      {best horizontal resolution; }
                           { $00480000 specifies 72 dpi}
   vRes:       Fixed;      {best vertical resolution; }
                           { $00480000 specifies 72 dpi}
   version:    Integer;    {set to -2}
   reserved1:  Integer;    {reserved; set to 0}
   reserved2:  LongInt;    {reserved; set to 0}
END;

CommentSpecHandle = ^CommentSpecPtr;
CommentSpecPtr    = ^CommentSpec;
CommentSpec       =     {comment specification record}
RECORD
   count:   Integer;    {number of times this type of comment }
                        { occurs in the picture or survey}
   ID:      Integer;    {value identifying this type of comment}
END;

FontSpecHandle = ^FontSpecPtr;
FontSpecPtr    = ^FontSpec;
FontSpec       =           {font specification record}
RECORD
   pictFontID: Integer;    {font ID as stored in the picture}
   sysFontID:  Integer;    {font family ID}
   size:       ARRAY[0..3] OF LongInt;
                           {each bit set from 1 to 127 indicates a }
                           { point size at that value; if bit 0 is }
                           { set, then a size larger than 127 }
                           { points is found}
   style:      Integer;    {styles used for this font family}
   nameOffset: LongInt;    {offset to font name stored in the }
                           { data structure indicated by the }
                           { fontNamesHandle field of the PictInfo }
                           { record}
END;
```

```
PictInfoID     = LongInt;

PictInfoHandle = ^PictInfoPtr;
PictInfoPtr    = ^PictInfo;
PictInfo       =                        {picture information record}
RECORD
   version:          Integer;           {Picture Utilities version number}
   uniqueColors:     LongInt;           {total colors in survey}
   thePalette:       PaletteHandle;     {handle to a Palette record--NIL }
                                        { for a bitmap in a basic }
                                        { graphics port}
   theColorTable:    CTabHandle;        {handle to a ColorTable record-- }
                                        { NIL for a bitmap in a basic }
                                        { graphics port}
   hRes:             Fixed;             {best horizontal resolution (dpi)}
   vRes:             Fixed;             {best vertical resolution (dpi)}
   depth:            Integer;           {greatest pixel depth}
   sourceRect:       Rect;              {optimal bounding rectangle for }
                                        { picture for display at }
                                        { resolution specified in hRes }
                                        { and vRes fields}
   textCount:        LongInt;           {number of text strings in }
                                        { picture(s)}
   lineCount:        LongInt;           {number of lines in picture(s)}
   rectCount:        LongInt;           {number of rectangles in }
                                        { picture(s)}
   rRectCount:       LongInt;           {number of rounded rectangles in }
                                        { picture(s)}
   ovalCount:        LongInt;           {number of ovals in picture(s)}
   arcCount:         LongInt;           {number of arcs and wedges in }
                                        { picture(s)}
   polyCount:        LongInt;           {number of polygons in picture(s)}
   regionCount:      LongInt;           {number of regions in picture(s)}
   bitMapCount:      LongInt;           {number of bitmaps}
   pixMapCount:      LongInt;           {number of pixel maps}
   commentCount:     LongInt;           {number of comments in picture(s)}
   uniqueComments:   LongInt;           {number of different comments }
                                        { (by ID) in picture(s)}
   commentHandle:    CommentSpecHandle;
                                        {handle to array of CommentSpec }
                                        { records for picture(s)}
   uniqueFonts:      LongInt;           {number of fonts in picture(s)}
   fontHandle:       FontSpecHandle;    {handle to an array of FontSpec }
                                        { records for picture(s)}
```

```
    fontNamesHandle:  Handle;          {handle to list of font names for }
                                       { picture(s)}
    reserved1:        LongInt;
    reserved2:        LongInt;
  END;
```

Routines

Creating and Disposing of Pictures

```
FUNCTION OpenCPicture       (newHeader: OpenCPicParams): PicHandle;
FUNCTION OpenPicture        (picFrame: Rect): PicHandle;
PROCEDURE PicComment        (kind,dataSize: Integer; dataHandle: Handle);
PROCEDURE ClosePicture;
PROCEDURE KillPicture       (myPicture: PicHandle);
```

Drawing Pictures

```
PROCEDURE DrawPicture       (myPicture: PicHandle; dstRect: Rect);
FUNCTION GetPicture         (picID: Integer): PicHandle;
```

Collecting Picture Information

```
/* DisposePictInfo is also spelled as DisposPictInfo */
FUNCTION GetPictInfo        (thePictHandle: PicHandle;
                             VAR thePictInfo: PictInfo; verb: Integer;
                             colorsRequested: Integer;
                             colorPickMethod: Integer;
                             version: Integer): OSErr;
FUNCTION GetPixMapInfo      (thePixMapHandle: PixMapHandle;
                             VAR thePictInfo: PictInfo; verb: Integer;
                             colorsRequested: Integer;
                             colorPickMethod: Integer;
                             version: Integer): OSErr;
FUNCTION NewPictInfo        (VAR thePictInfoID: PictInfoID; verb: Integer;
                             colorsRequested: Integer;
                             colorPickMethod: Integer;
                             version: Integer): OSErr;
FUNCTION RecordPictInfo     (thePictInfoID: PictInfoID;
                             thePictHandle: PicHandle): OSErr;
```

```
FUNCTION RecordPixMapInfo   (thePictInfoID: PictInfoID;
                             thePixMapHandle: PixMapHandle): OSErr;
FUNCTION RetrievePictInfo   (thePictInfoID: PictInfoID;
                             VAR thePictInfo: PictInfo;
                             colorsRequested: Integer): OSErr;
FUNCTION DisposePictInfo    (thePictInfoID: PictInfoID): OSErr;
```

Application-Defined Routines

```
FUNCTION MyInitPickMethod   (colorsRequested: Integer;
                             VAR dataRef: LongInt;
                             VAR colorBankType: Integer): OSErr;
FUNCTION MyRecordColors     (dataRef: LongInt; colorsArray: RGBColorArray;
                             colorCount: LongInt;
                             VAR uniqueColors: LongInt): OSErr;
FUNCTION MyCalcColorTable   (dataRef: LongInt; colorsRequested: Integer;
                             colorBankPtr: Ptr;
                             VAR resultPtr: CSpecArray): OSErr;
FUNCTION MyDisposeColorPickMethod
                            (dataRef: LongInt): OSErr;
```

C Summary

Constants

```
/* color-picking methods */
#define systemMethod    0  /* let Picture Utilities choose the method
                              (currently they always choose popularMethod) */
#define popularMethod   1  /* return most frequently used colors */
#define medianMethod    2  /* return a weighted distribution of colors */

/* picture information to be returned by Picture Utilities */
#define returnColorTable   ((short) 0x0001)  /* return a ColorTable record */
#define returnPalette      ((short) 0x0002)  /* return a Palette record */
#define recordComments     ((short) 0x0004)  /* return comment information */
#define recordFontInfo     ((short) 0x0008)  /* return font information */
#define suppressBlackAndWhite
                           ((short) 0x0010)  /* don't include black and
                                                white with returned colors */
```

```
/* color bank types */
#define ColorBankIsCustom        -1    /* gathers colors into a custom
                                          color bank */
#define ColorBankIsExactAnd555   0     /* gathers exact colors if there are
                                          less than 256 unique colors in
                                          picture; otherwise gathers colors
                                          for picture in a 5-5-5 histogram */
#define ColorBankIs555           1     /* gathers colors in a 5-5-5
                                          histogram */
```

Data Types

```
struct Picture {
   short    picSize;    /* for a version 1 picture: its size */
   Rect     picFrame;   /* bounding rectangle for the picture */
   /* variable amount of picture data in the form of opcodes */
};
typedef struct Picture Picture;
typedef Picture *PicPtr, **PicHandle;

struct OpenCPicParams {
   Rect  srcRect;    /* optimal bounding rectangle for displaying picture at
                        resolution indicated in hRes, vRes fields */
   Fixed hRes;       /* best horizontal resolution; $00480000 specifies
                        72 dpi */
   Fixed vRes;       /* best vertical resolution; $00480000 specifies
                        72 dpi */
   short version;    /* set to -2 */
   short reserved1;  /* reserved; set to 0 */
   long  reserved2;  /* reserved; set to 0 */
};

struct CommentSpec {
   short    count;   /* number of times this type of comment occurs in
                        the picture or survey */
   short    ID;      /* value identifying this type of comment */
};

typedef struct CommentSpec CommentSpec;
typedef CommentSpec *CommentSpecPtr, **CommentSpecHandle;
```

```
struct FontSpec {       /* font specification record */
   short    pictFontID; /* font ID as stored in the picture */
   short    sysFontID;  /* font family ID */
   long     size[4];    /* each bit set from 1 to 127 indicates a point
                           size at that value; if bit 0 is set, then a size
                           larger than 127 is found */
   short    style;      /* styles used for this font family */
   long     nameOffset; /* offset to font name stored in the data structure
                           indicated by the fontNamesHandle field of the
                           PictInfo record */
};
typedef struct FontSpec FontSpec;
typedef FontSpec *FontSpecPtr, **FontSpecHandle;

struct  PictInfo {
   short             version;       /* Picture Utilities version number */
   long              uniqueColors;  /* total colors in survey */
   PaletteHandle     thePalette;    /* handle to a Palette record--NIL for
                                       a bitmap in a basic graphics port */
   CTabHandle        theColorTable; /* handle to a ColorTable record--NIL for
                                       a bitmap in a basic graphics port */
   Fixed             hRes;          /* best horizontal resolution (dpi)} */
   Fixed             vRes;          /* best vertical resolution (dpi) */
   short             depth;         /* greatest pixel depth */
   Rect              sourceRect;    /* optimal bounding rectangle for
                                       picture for display at resolution
                                       specified in hRes and vRes fields */
   long              textCount;     /* number of text strings in
                                       picture(s) */
   long              lineCount;     /* number of lines in picture(s) */
   long              rectCount;     /* number of rectangles in picture(s) */
   long              rRectCount;    /* number of rounded rectangles in
                                       picture(s) */
   long              ovalCount;     /* number of ovals in picture(s) */
   long              arcCount;      /* number of arcs and wedges in
                                       picture(s) */
   long              polyCount;     /* number of polygons in picture(s) */
   long              regionCount;   /* number of regions in picture(s) */
   long              bitMapCount;   /* number of bitmaps */
   long              pixMapCount;   /* number of pixel maps */
   long              commentCount;  /* number of comments in picture(s) */
   long              uniqueComments;
                                    /* number of different comments (by ID)
                                       in picture(s) */
```

```
   CommentSpecHandle commentHandle; /* handle to an array of CommentSpec
                                       structures for picture(s) */
   long              uniqueFonts;   /* number of fonts in picture(s) */
   FontSpecHandle    fontHandle;    /* handle to an array of FontSpec
                                       structures for picture(s) */
   Handle            fontNamesHandle;
                                    /* handle to list of font names for
                                       picture(s) */
   long              reserved1;
   long              reserved2;
};
typedef struct PictInfo PictInfo;
typedef PictInfo *PictInfoPtr,**PictInfoHandle;

typedef long PictInfoID;
```

Functions

Creating and Disposing of Pictures

```
pascal PicHandle OpenCPicture
                           (const OpenCPicParams *newHeader);
pascal PicHandle OpenPicture
                           (const Rect *picFrame);
pascal void PicComment     (short kind, short dataSize, Handle dataHandle);
pascal void ClosePicture   (void);
pascal void KillPicture    (PicHandle myPicture);
```

Drawing Pictures

```
pascal void DrawPicture    (PicHandle myPicture, const Rect *dstRect);
pascal PicHandle GetPicture (Integer picID);
```

Collecting Picture Information

```
pascal OSErr GetPictInfo    (PicHandle thePictHandle,
                             PictInfo *thePictInfo, short verb,
                             short colorsRequested, short colorPickMethod,
                             short version);
pascal OSErr GetPixMapInfo  (PixMapHandle thePixMapHandle,
                             pictInfo *thePictInfo, short verb,
                             short colorsRequested, short colorPickMethod,
                             short version);
pascal OSErr NewPictInfo    (PictInfoID *thePictInfoID, short verb,
                             short colorsRequested, short colorPickMethod,
                             short version);
pascal OSErr RecordPictInfo (PictInfoID thePictInfoID,
                             PicHandle thePictHandle);
pascal OSErr RecordPixMapInfo
                            (PictInfoID thePictInfoID,
                             PixMapHandle thePixMapHandle);
pascal OSErr RetrievePictInfo
                            (PictInfoID thePictInfoID,
                             PictInfo *thePictInfo, short colorsRequested);
pascal OSErr DisposePictInfo
                            (PictInfoID thePictInfoID);
```

Application-Defined Functions

```
pascal OSErr MyInitPickMethod
                            (short colorsRequested, long *dataRef,
                             short *colorBankType);
pascal OSErr MyRecordColors (long dataRef, RGBColorArray colorsArray,
                             long colorCount, long *uniqueColors);
pascal OSErr MyCalcColorTable
                            (long dataRef, short colorsRequested,
                             Ptr colorBankPtr, CSpecArray *resultPtr);
pascal OSErr MyDisposeColorPickMethod
                            (long dataRef);
```

Assembly-Language Summary

Data Structures

Picture Data Structure

0	`picSize`	word	for a version 1 picture: its size
2	`picFrame`	8 bytes	bounding rectangle for the picture
10	`picData`	variable	variable amount of picture data

OpenCPicParams Data Structure

0	`srcRect`	8 bytes	optimal bounding rectangle for displaying picture at `hRes`, `vRes`
8	`hRes`	long	best horizontal resolution
12	`vRes`	long	best vertical resolution
16	`version`	word	always set to –2
18	`reserved1`	word	reserved; set to 0
20	`reserved2`	long	reserved; set to 0

CommentSpec Data Structure

0	`count`	long	number of times this type of comment occurs in picture or survey
4	`ID`	long	value identifying this type of comment

FontSpec Data Structure

0	`pictFontID`	word	font ID as stored in the picture
2	`sysFontID`	word	font family ID
4	`size`	8 bytes	each bit set from 1 to 127 indicates a point size at that value; if bit 0 is set, then a size larger than 127 points is found
12	`style`	word	styles used for this font family
14	`nameOffset`	long	offset to font name stored in the data structure indicated by the `fontNamesHandle` field of the `PictInfo` record

PictInfo Data Structure

0	`version`	word	Picture Utilities version number
2	`uniqueColors`	long	total number of colors in survey
6	`thePalette`	long	handle to a Palette record—`NIL` for a bitmap in a basic graphics port
10	`theColorTable`	long	handle to a `ColorTable` record—`NIL` for a bitmap in a basic graphics port
14	`hRes`	long	best horizontal resolution (dpi)
18	`vRes`	long	best vertical resolution (dpi)
22	`depth`	word	greatest pixel depth
24	`sourceRect`	8 bytes	optimal bounding rectangle for picture for display at resolution specified in `hRes` and `vRes` fields
32	`textCount`	long	number of text strings in picture(s)
36	`lineCount`	long	number of lines in picture(s)
40	`rectCount`	long	number of rectangles in picture(s)
44	`rRectCount`	long	number of rounded rectangles in picture(s)
48	`ovalCount`	long	number of ovals in picture(s)
52	`arcCount`	long	number of arcs and wedges in picture(s)
56	`polyCount`	long	number of polygons in picture(s)
60	`regionCount`	long	number of regions in picture(s)
64	`bitMapCount`	long	number of bitmaps
68	`pixMapCount`	long	number of pixel maps
72	`commentCount`	long	number of comments in picture(s)
76	`uniqueComments`	long	number of different comments (by ID) in picture(s)
80	`commentHandle`	long	handle to an array of `CommentSpec` records for picture(s)
84	`uniqueFonts`	long	number of fonts in picture(s)
88	`fontHandle`	long	handle to an array of `FontSpec` records for picture(s)
92	`fontNamesHandle`	long	handle to list of font names for picture(s)
96	`reserved1`	long	reserved
100	`reserved2`	long	reserved

Trap Macros

Trap Macros Requiring Routine Selectors

`_Pack15`

Selector	Routine
$0206	`DisposePictInfo`
$0403	`RecordPictInfo`
$0404	`RecordPixMapInfo`
$0505	`RetrievePictInfo`
$0602	`NewPictInfo`
$0800	`GetPictInfo`
$0801	`GetPixMapInfo`

Result Codes

`pictInfoVersionErr`	–11000	Version number not 0
`pictInfoIDErr`	–11001	Invalid picture information ID
`pictInfoVerbErr`	–11002	Invalid verb combination specified
`cantLoadPickMethodErr`	–11003	Custom pick method not in resource chain
`colorsRequestedErr`	–11004	Number out of range or greater than that passed to `NewPictInfo`
`pictureDataErr`	–11005	Invalid picture data

Cursor Utilities

Contents

Errata for Chapter 8: Cursor Utilities 8-2
About the Cursor 8-3
Using the Cursor Utilities 8-5
  Initializing the Cursor 8-6
  Changing the Appearance of the Cursor 8-7
  Creating an Animated Cursor 8-13
Cursor Utilities Reference 8-16
  Data Structures 8-16
  Routines 8-21
    Initializing Cursors 8-21
    Changing Black-and-White Cursors 8-24
    Changing Color Cursors 8-25
    Hiding and Showing Cursors 8-28
    Displaying Animated Cursors 8-31
  Resources 8-33
    The Cursor Resource 8-33
    The Color Cursor Resource 8-34
    The Animated Cursor Resource 8-36
Summary of Cursor Utilities 8-38
  Pascal Summary 8-38
    Constants 8-38
    Data Types 8-38
    Routines 8-39
  C Summary 8-40
    Constants 8-40
    Data Types 8-41
    Functions 8-42
  Assembly-Language Summary 8-43
    Data Structures 8-43
    Global Variables 8-43

Errata for Chapter 8: Cursor Utilities

This chapter describes how to use the following functions to animate cursors:

- `InitCursorCtl`
- `Hide_Cursor`
- `Show_Cursor`
- `RotateCursor`
- `SpinCursor`

These functions are available in the MPW programming environment and possibly other environments, but they are not supported by the Mac OS. Moreover, these functions, which are defined in the MPW interface file `CursorCtl.h`, are not supported by Carbon.

This chapter describes the utilities that your application uses to draw and manipulate the cursor on the screen. You should read this chapter to find out how to implement cursors in your application. For example, you should change the arrow cursor to an I-beam cursor when it's over text and to an animated cursor when a medium-length process is under way.

Cursors are defined in resources; the routines in this chapter automatically call the Resource Manager as necessary. For more information about resources, see the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*. Color cursors are defined in resources as well, though they use Color QuickDraw. For information about Color QuickDraw, see the chapter "Color QuickDraw" in this book.

This chapter describes how to

- create and display black-and-white and color cursors
- change the cursor's shape over different areas on the screen
- display an animated cursor

About the Cursor

A **cursor** is a 256-pixel, black-and-white image in a 16-by-16 pixel square usually defined by an application in a cursor (`'CURS'`) resource. The cursor is an integral part of the Macintosh user interface. The user manipulates the cursor with the mouse to select objects or areas on the screen. (It appears only on the screen and never in an offscreen graphics port.) The user moves the cursor across the screen by moving the mouse. Most actions take place only when the user positions the cursor over an object on the screen, then clicks (presses and releases the mouse button). For example, a user might point at a document icon created by your application and click to select it, then choose the Open command from the File menu by pointing at it with the mouse button depressed and then releasing the mouse button.

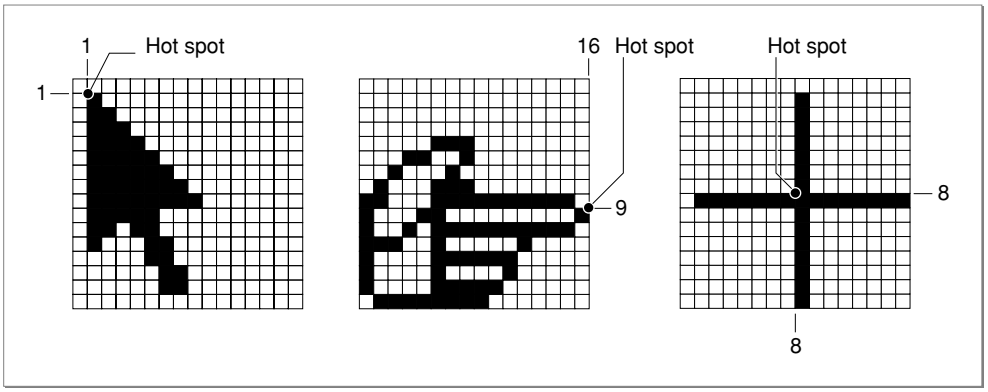
You use a cursor in the content area of your application's windows to allow the user to select all or part of the content. Your application also uses the cursor in the scroll bar area of its windows to adjust the position of the document's contents in the window area. You can change the shape of the cursor to indicate that a user is over a certain kind of content, such as text, or to provide feedback about the status of the computer system.

Note
Some Macintosh user manuals call the cursor a *pointer* because it points to a location on the screen. To avoid confusion with other meanings of *pointer*, *Inside Macintosh* uses the alternate term *cursor*. ◆

Basic QuickDraw supplies a predefined cursor in the global variable named `arrow`; this is the standard arrow cursor.

One point in the cursor's image is designated as the **hot spot**, which in turn points to a location on the screen. The hot spot is the portion of the pointer that must be positioned over a screen object before mouse clicks can have an effect on that object. For example, when the user presses the mouse button, the Event Manager function `WaitNextEvent` reports the location of the cursor's hot spot in global coordinates. Figure 8-1 illustrates three cursors and their hot spot points.

Figure 8-1 Hot spots in cursors



The hot spot is a point (not a bit) in the bit image for the cursor. Imagine the rectangle with corners (0,0) and (16,16) containing the cursor's bit image, as in each of the examples in Figure 8-1; each hot spot is defined in the local coordinate systems of these rectangles. For the arrow cursor in this figure, local coordinates (1,1) designate the hot spot. A hot spot of (8,8) is in the center of the crosshairs cursor in Figure 8-1. Notice that the hot spot for the pointing hand cursor has a horizontal coordinate of 16 and a vertical coordinate of 9.

Whenever the user moves the mouse, the low-level interrupt-driven mouse routines move the cursor to a new location on the screen. Your application doesn't need to do anything to move the cursor.

Your application should change the cursor shape depending on where the user positions it on the screen. For example, when the cursor is in your application's menu bar, the cursor should usually have an arrow shape. When the user moves the cursor over a text document, your application should change the cursor's shape to an I-beam, which indicates where the insertion point will move if the user clicks. When it's over graphic objects, the cursor may have different shapes depending on the type of graphic and the operation that the user is attempting to complete. You should change the cursor shape *only* to provide information to the user. In other words, don't change its shape randomly.

In general, you should always make the cursor visible in your application. To maintain a stable and consistent environment, the user should have access to the cursor. There are a few cases when the cursor may not be visible. For example, in an application where the user is entering text, the insertion point should blink and the cursor should not be visible. If the cursor and the insertion point were both visible, it might confuse the user about where the input would appear. Or, if the user is viewing a slide show in a presentation software application, the cursor need not be visible. However, whenever the user needs access to the cursor, a simple move of the mouse should make the cursor visible again.

When the cursor is used for choosing and selecting, it should remain black. You may want to display a color cursor when the user is drawing or typing in color. The cursor shouldn't contain more than one color at a time, with the exception of a multicolored paintbrush cursor. It's hard for the eye to distinguish small areas of color. Make sure that the hot spot can be seen when it's placed on a background of a similar color. This can be accomplished by changing the color of the cursor or by adding a one-pixel outline in a contrasting color.

When your application is performing an operation that will take at least a couple of seconds, and more time than a user might expect, you need to provide feedback to the user that the operation is in progress. If the operation will last a second or two (a short operation), change the cursor to the wristwatch cursor. If the operation takes several seconds (a medium-length operation) and the user can do nothing in your application but stop the operation, wait until it is completed, or switch to another application, you need to display an animated cursor. This lets the user know that the computer system hasn't crashed—it's just busy. If the operation will take longer than several seconds (a lengthy operation), your application should display a status indicator to show the user the estimated total time and the elapsing time of the operation.

For more information about displaying cursors and status indicators in your application, see *Macintosh Human Interface Guidelines*.

Using the Cursor Utilities

This section describes how you can

- create cursors
- change the shape of the cursor
- animate a cursor to indicate that a medium-length process is taking place

To implement cursors, you need to

- define black-and-white cursors as `'CURS'` resources in the resource file of your application
- define color cursors in `'crsr'` resources—if you want to display color cursors—in the resource file of your application
- define `'acur'` resources—if you want to display animated cursors—in the resource file of your application
- initialize the Cursor Utilities by using the `InitCursor` and `InitCursorCtl` procedures when your application starts up
- use the `SetCursor` or `SetCCursor` procedure to change the cursor shape as necessary
- animate the cursor by using the `SpinCursor` or `RotateCursor` procedure

You use `'CURS'` resources to create black-and-white cursors for display on black-and-white and color screens. You use `'crsr'` resources to create color cursors for display on systems supporting Color QuickDraw. Each `'crsr'` resource also contains a black-and-white image that Color QuickDraw displays on black-and-white screens.

Before using the routines that handle color cursors—namely, the `GetCCursor`, `SetCCursor`, and `DisposeCCursor` routines—you must test for the existence of Color QuickDraw by using the `Gestalt` function with the `GestaltQuickDrawVersion` selector. If the value returned in the `response` parameter is equal to or greater than the value of the constant `gestalt32BitQD`, then the system supports Color QuickDraw. Both basic and Color QuickDraw support all other routines described in this chapter.

Initializing the Cursor

When your application starts up, the Finder sets the cursor to a wristwatch; this indicates that an operation is in progress. When your application nears completion of its initialization tasks, it should call the `InitCursor` procedure to change the cursor from a wristwatch to an arrow, as shown in the application-defined procedure `DoInit` in Listing 8-1.

Listing 8-1 Initializing the Cursor Utilities

```
PROCEDURE DoInit;
BEGIN
   DoSetUpHeap;       {perform Memory Manager initialization here}
   InitGraf(@thePort);{initialize basic 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 }
                      { wristwatch}
   InitCursorCtl(NIL);{load resources for animated cursor with }
                      { resource ID 0}
END; {of DoInit}
```

If your application uses an animated cursor to indicate that an operation of medium length is under way, it should also call the `InitCursorCtl` procedure to load its `'acur'` resource and associated `'CURS'` resources, as illustrated in Listing 8-1.

Changing the Appearance of the Cursor

Whenever the user moves the mouse, the mouse driver, the Event Manager, and your application are responsible for providing feedback to the user. The mouse driver performs low-level functions, such as continually polling the mouse for its location and status and maintaining the current location of the mouse in a global variable. Whenever the user moves the mouse, a low-level interrupt routine of the mouse driver moves the cursor displayed on the screen and aligns the hot spot of the cursor with the new mouse location. This section describes how to use the `GetCursor` and `SetCursor` routines to change the appearance of a black-and-white cursor when it is in different areas of the screen. (To change the cursor to a color cursor, your application must use the `GetCCursor` function, described on page 8-26, and the `SetCCursor` procedure, described on page 8-26.)

Your application is responsible for setting the initial appearance of the cursor, for restoring the cursor after the Event Manager function `WaitNextEvent` returns, and for changing the appearance of the cursor as appropriate for your application. For example, most applications set the cursor to the I-beam when the cursor is inside a text-editing area of a document, and they change the cursor to an arrow when the cursor is inside a scroll bar of a document. Your application can achieve this effect by requesting that the Event Manager report mouse-moved events if the user moves the cursor out of a region you specify in the `mouseRgn` parameter to the `WaitNextEvent` function. `WaitNextEvent` is described in the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

The mouse driver and your application control the shape and appearance of the cursor. A cursor can be any 256-pixel image, defined by a 16-by-16 pixel square. The mouse driver displays the current cursor, which your application can change by using the `SetCursor` or `SetCCursor` procedure.

Figure 8-2 shows the standard arrow cursor. You initialize the cursor to the standard arrow cursor when you use the `InitCursor` procedure, as shown in Listing 8-1. As shown in Figure 8-2, the hot spot for the arrow cursor is at location (1,1).

Figure 8-2 The standard arrow cursor

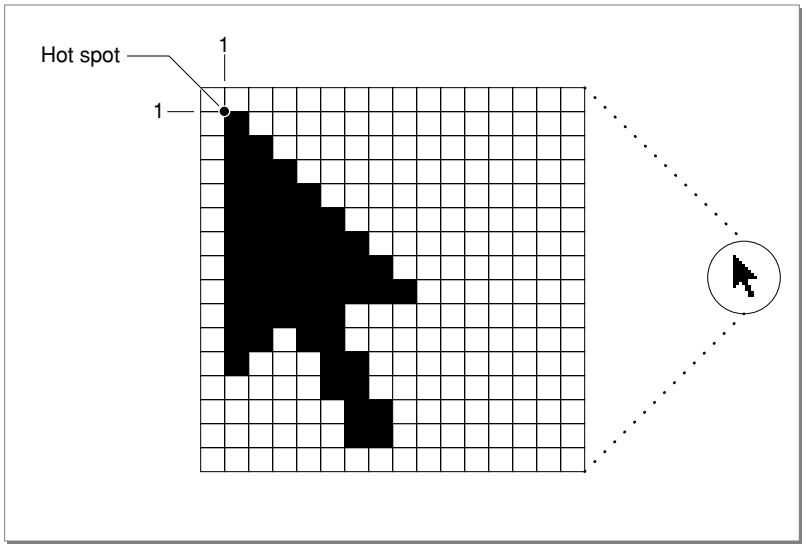
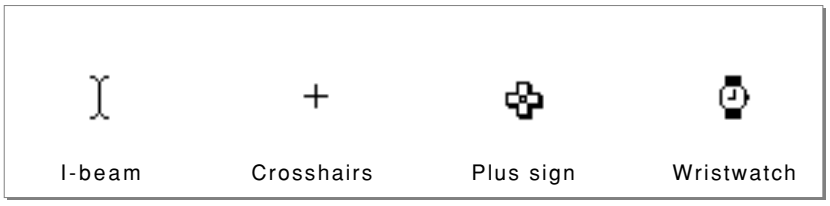


Figure 8-3 shows four other common cursors that are available to your application: the I-beam, crosshairs, plus sign, and wristwatch cursors.

Figure 8-3 The I-beam, crosshairs, plus sign, and wristwatch cursors



The I-beam, crosshairs, plus sign, and wristwatch cursors are defined as resources, and your application can get a handle to any of these cursors by specifying their corresponding resource IDs to the `GetCursor` function. These constants specify the resource IDs for these common cursors:

```
CONST iBeamCursor = 1;  {used in text editing}
      crossCursor = 2;  {often used for manipulating graphics}
      plusCursor  = 3;  {often used for selecting fields in }
                        { an array}
```

```
watchCursor = 4;  {used when a short operation is in }
                  { progress}
```

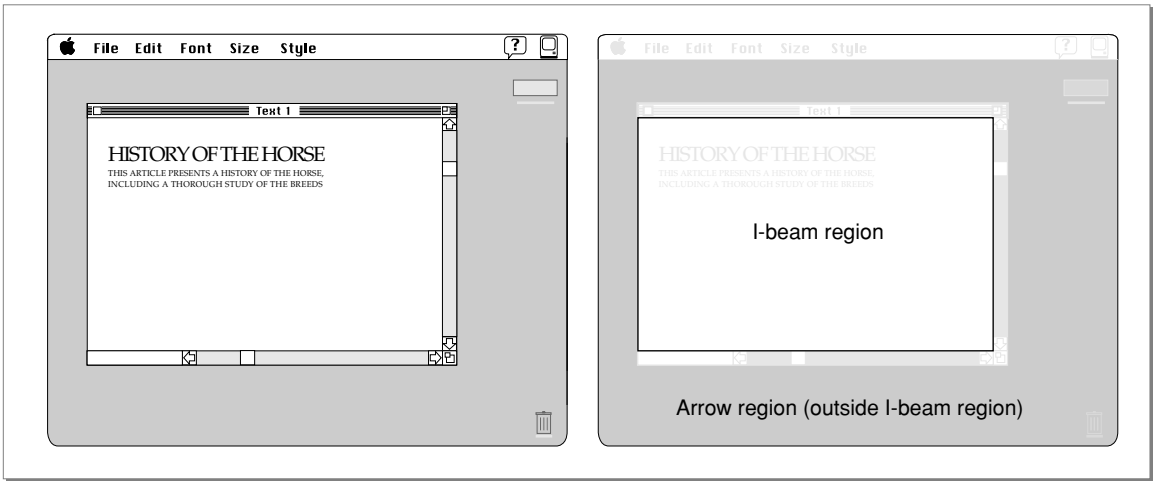
After you use the `GetCursor` function to obtain a handle to one of these cursors or to one defined by your own application in a `'CURS'` resource, you can change the appearance of the cursor by using the `SetCursor` procedure.

Your application usually needs to change the shape of the cursor as the user moves the cursor to different areas within a document. Your application can use mouse-moved events to help accomplish this. Your application also needs to adjust the cursor in response to resume events. Most applications adjust the cursor once through the event loop in response to almost all events.

You can request that the Event Manager report mouse-moved events whenever the cursor is outside of a specified region that you pass as a parameter to the `WaitNextEvent` function. (If you specify an empty region or a `NIL` handle to the `WaitNextEvent` function, `WaitNextEvent` does not report mouse-moved events.)

If you specify a nonempty region in the `mouseRgn` parameter to the `WaitNextEvent` function, `WaitNextEvent` returns a mouse-moved event whenever the cursor is outside of that region. For example, Figure 8-4 shows a document window. Your application might define two regions: a region that encloses the text area of the window (the I-beam region), and a region that defines the scroll bars and all other areas outside the text area (the arrow region). If your application has specified the I-beam region to `WaitNextEvent`, the mouse driver continues to display the I-beam cursor until the user moves the cursor out of the region.

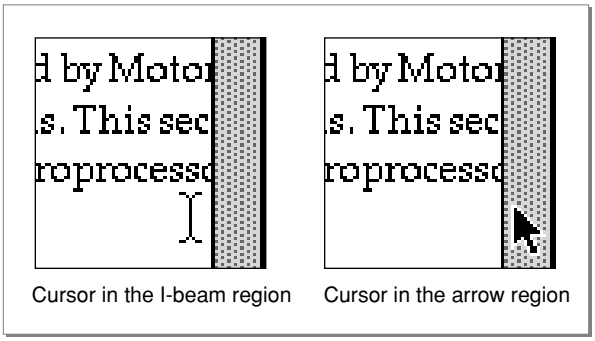
Figure 8-4 A window and its arrow and I-beam regions



When the user moves the cursor out of the I-beam region, `WaitNextEvent` reports a mouse-moved event. Your application can then change the I-beam cursor to the arrow cursor and change the `mouseRgn` parameter to the area defined by the scroll bars and all other areas outside of the I-beam region. The cursor remains an arrow until the user moves the cursor out of the arrow region, at which point your application receives a mouse-moved event.

Figure 8-5 shows how an application might change the cursor from the I-beam cursor to the arrow cursor after receiving a mouse-moved event.

Figure 8-5 Changing the cursor from the I-beam cursor to the arrow cursor



Note that your application should recalculate the `mouseRgn` parameter when it receives a mouse-moved event; otherwise, it will continue to receive mouse-moved events as long as the cursor position is outside the original region.

Listing 8-2 shows an application-defined routine called `MyAdjustCursor`. After receiving any event other than a high-level event, the application's event loop (described in the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*) calls `MyAdjustCursor` to adjust the cursor.

Listing 8-2 Changing the cursor

```
PROCEDURE MyAdjustCursor (mouse: Point; VAR region: RgnHandle);
VAR
   window:        WindowPtr;
   arrowRgn:      RgnHandle;
   iBeamRgn:      RgnHandle;
   iBeamRect:     Rect;
   myData:        MyDocRecHnd;
   windowType:    Integer;
BEGIN
window := FrontWindow;
{Determine the type of window--document, modeless, etc.}
```

```
windowType := MyGetWindowType(window);
CASE windowType OF
   kMyDocWindow:
   BEGIN
      {initialize regions for arrow and I-beam}
      arrowRgn := NewRgn;
      ibeamRgn := NewRgn;
      {set arrow region to large region at first}
      SetRectRgn(arrowRgn, -32768, -32768, 32766, 32766);
      {calculate I-beam region}
      {first get the document's TextEdit view rectangle}
      myData := MyDocRecHnd(GetWRefCon(window));
      iBeamRect := myData^^.editRec^^.viewRect;
      SetPort(window);
      WITH iBeamRect DO
      BEGIN
         LocalToGlobal(topLeft);
         LocalToGlobal(botRight);
      END;
      RectRgn(iBeamRgn, iBeamRect);
      WITH window^.portBits.bounds DO
         SetOrigin(-left, -top);
      {intersect I-beam region with window's visible region}
      SectRgn(iBeamRgn, window^.visRgn, iBeamRgn);
      SetOrigin(0,0);
      {calculate arrow region by subtracting I-beam region}
      DiffRgn(arrowRgn, iBeamRgn, arrowRgn);
      {change the cursor and region parameter as necessary}
      IF PtInRgn(mouse, iBeamRgn) THEN {cursor is in I-beam rgn}
      BEGIN
         SetCursor(GetCursor(iBeamCursor)^^);      {set to I-beam}
         CopyRgn(iBeamRgn, region);     {update the region param}
      END;
      {update cursor if in arrow region}
      IF PtInRgn(mouse, arrowRgn) THEN {cursor is in arrow rgn}
      BEGIN
         SetCursor(arrow);             {set cursor to the arrow}
         CopyRgn(arrowRgn, region);    {update the region param}
      END;
      DisposeRgn(iBeamRgn);
      DisposeRgn(arrowRgn);
   END; {of kMyDocWindow}
```

```
   kMyGlobalChangesID:
      MyCalcCursorRgnForModelessDialogBox(window, region);
   kNil:
   BEGIN
      MySetRegionNoWindows(kNil, region);
      SetCursor(arrow);
   END;
   END; {of CASE}
END;
```

The `MyAdjustCursor` procedure sets the cursor appropriately, according to whether a document window or modeless dialog box is active.

For a document window, `MyAdjustCursor` defines two regions, specified by the `arrowRgn` and `iBeamRgn` variables. If the cursor is inside the region described by the `arrowRgn` variable, `MyAdjustCursor` sets the cursor to the arrow cursor and returns the region described by `arrowRgn`. Similarly, if the cursor is inside the region described by the `iBeamRgn` variable, `MyAdjustCursor` sets the cursor to the I-beam cursor and returns the region described by `iBeamRgn`.

The `MyAdjustCursor` procedure calculates the two regions by first setting the arrow region to the largest possible region. It then sets the I-beam region to the region described by the document's TextEdit view rectangle. This region typically corresponds to the content area of the window minus the scroll bars. (If your application doesn't use TextEdit for its document window, then set this region as appropriate to your application.) The `MyAdjustCursor` routine adjusts the I-beam region so that it includes only the part of the content area that is in the window's visible region (for example, to take into account any floating windows that might be over the window). The code in this listing sets the arrow region to include the entire screen except for the region occupied by the I-beam region. (TextEdit is described in *Inside Macintosh: Text*.)

The `MyAdjustCursor` procedure then determines which region the cursor is in and sets the cursor and region parameter appropriately.

For modeless dialog boxes, `MyAdjustCursor` calls its own routine to appropriately adjust the cursor for the modeless dialog box. The `MyAdjustCursor` procedure also appropriately adjusts the cursor if no windows are currently open.

Your application should normally hide the cursor when the user is typing. You can remove the cursor image from the screen by using either the `HideCursor` or `Hide_Cursor` procedure. You can hide the cursor temporarily by using the `ObscureCursor` procedure, or you can hide the cursor in a given rectangle by using the `ShieldCursor` procedure. To display a hidden cursor, use the `ShowCursor` or `Show_Cursor` procedure. Note that you do not need to explicitly show the cursor after

your application uses the `ObscureCursor` procedure; instead, the cursor automatically reappears when the user moves the mouse again. These procedures are described in "Hiding and Showing Cursors" beginning on page 8-28.

Creating an Animated Cursor

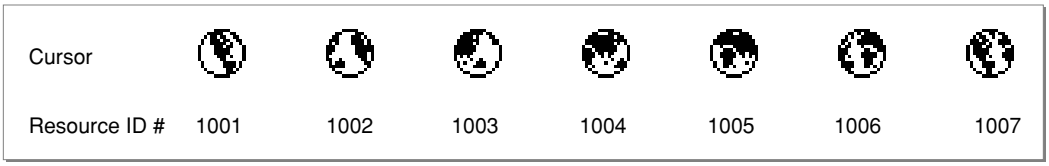
Your application should display an animated cursor when performing a medium-length operation that might cause the user to think that the computer has stopped working. To create an animated cursor, you should

- create a series of `'CURS'` resources that make up the "frames" of the animation
- create an `'acur'` resource with a resource ID of 0
- pass the value `NIL` to the `InitCursorCtl` procedure once in your program code to load these resources
- use either the `RotateCursor` or `SpinCursor` procedure when your application is busy with its task

Note
An alternate, but more code-intensive, method of creating and displaying an animated cursor is shown in the chapter "Vertical Retrace Manager" in *Inside Macintosh: Processes*. ◆

Typically, an animated cursor uses four to seven frames. For example, the seven `'CURS'` resources in Figure 8-6 constitute the seven frames of a globe cursor that spins. To create these resources, your application typically uses a high-level utility such as ResEdit, which is available from APDA.

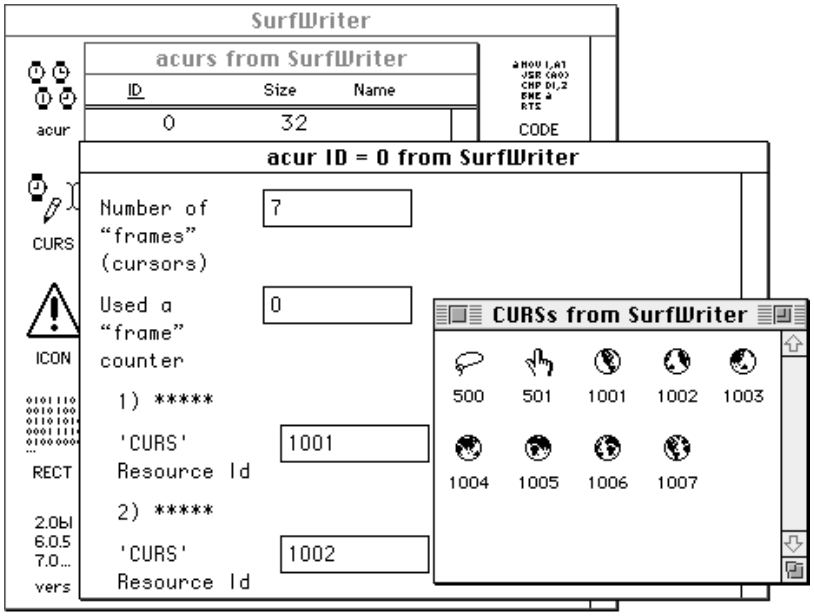
Figure 8-6 The `'CURS'` resources for an animated globe cursor



To collect and order your `'CURS'` frames into a single animation, you must create an `'acur'` resource. This resource specifies the IDs of the `'CURS'` resources and the sequence for displaying them in your animation. If your application uses only one spinning cursor, give your `'acur'` resource a resource ID of 0.

Figure 8-7 shows how the `'CURS'` resources for the spinning globe cursor are specified in an `'acur'` resource using ResEdit.

Figure 8-7 An `'acur'` resource for an animated cursor



To load the `'acur'` resource and its associated `'CURS'` resources, use the `InitCursorCtl` procedure once prior to calling the `RotateCursor` or `SpinCursor` procedure. If you pass `NIL` to `InitCursorCtl`, then it automatically loads the `'acur'` resource that has an ID of 0 in your application's resource file. If you wish to use multiple animated cursors, you must create multiple `'acur'` resources—that is, one for each series of `'CURS'` resources. Prior to displaying one of your animated cursors with `RotateCursor` or `SpinCursor`, you must call the Resource Manager function `GetResource` to return a handle to its `'acur'` resource. Your application must coerce that handle to one of type `acurHandle`, and then pass this handle to the `InitCursorCtl` procedure. See the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information about `GetResource`.

When you call `RotateCursor` or `SpinCursor`, one frame—that is, one `'CURS'` resource—is displayed. When you pass a positive value to the procedure the next time you call it, the next frame specified in the `'acur'` resource is displayed. A negative value passed to either procedure displays the previous frame listed in the `'acur'` resource. The distinction between `RotateCursor` and `SpinCursor` is that your application maintains an index for changing the cursor when calling `RotateCursor`, but your application does not maintain an index for changing the cursor when calling `SpinCursor`; instead, your application must determine the proper interval for calling `SpinCursor`.

Listing 8-3 shows an application-defined routine called `MyRotateCursor`. When the application calling `MyRotateCursor` starts on a medium-length operation and needs to indicate to the user that the operation is in progress, the application sets its global variable `gDone` to `FALSE` and repeatedly calls `MyRotateCursor` until the operation is complete and `gDone` becomes `TRUE`.

Listing 8-3 Animating a cursor with the `RotateCursor` procedure

```
PROCEDURE MyRotateCursor;
BEGIN
   IF NOT gDone THEN
   BEGIN
      RotateCursor(TickCount);
   END;
END;
```

Listing 8-3 uses the Event Manager function `TickCount` to maintain an index for `RotateCursor` to use when displaying the frames for an animated cursor. (A tick is approximately 1/60 of a second; `TickCount` returns the number of ticks since the computer started up.) When the value passed as a parameter to `RotateCursor` is a multiple of 32, then `RotateCursor` displays the next frame in the animation.

Listing 8-4 shows an application-defined routine called `MySpinCursor`. As you see in Listing 8-4, the application does not maintain an index for displaying the frames for an animated cursor. Instead, every time `SpinCursor` is called, the next frame in the animation is displayed.

Listing 8-4 Animating a cursor with the `SpinCursor` procedure

```
PROCEDURE MySpinCursor;
BEGIN
   IF NOT gDone THEN
      SpinCursor(0);
   END;
```

If the operation takes less than a second or two, your application can simply use the `SetCursor` procedure to display the cursor with the resource ID represented by the `watchCursor` constant. If the operation will take longer than several seconds (a lengthy operation), your application should display a status indicator in a dialog box to show the user the estimated total time and the elapsing time of the operation. See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information about creating and displaying dialog boxes.

Cursor Utilities Reference

This section describes the data structures, routines, and resources that are specific to cursors. "Data Structures" shows the Pascal data structures for the `Bits16` array and the `Cursor`, `CCrsr`, `Cursors`, and `Acur` records. "Routines" describes the routines for initializing cursors, managing black-and-white cursors, managing color cursors, hiding and showing cursors, and displaying animated cursors. "Resources" describes the cursor resource, the color cursor resource, and the animated cursor resource. The constants that represent values for the standard cursors are listed in "Summary of Cursor Utilities."

Data Structures

Your application typically does not create the data structures described in this section. Although you can create a `Cursor` record and its associated `Bits16` array in your program code, it is usually easier to create a black-and-white cursor in a cursor resource, which is described on page 8-33. Similarly, you can create a `CCrsr` record in your program code, but it is usually easier to create a color cursor in a color cursor resource, which is described on page 8-34. The `Cursors` data type contains the standard cursors you can display. Finally, you usually list animated cursors in an animated cursor resource, which is described on page 8-36, instead of creating them in an `Acur` record.

Bits16

The `Bits16` array is used by the `Cursor` record to hold a black-and-white, 16-by-16 pixel square image.

```
Bits16 = ARRAY[0..15] OF Integer;
```

Cursor

Your application typically does not create `Cursor` records, which are data structures of type `Cursor`. Although you can create a `Cursor` record and its associated `Bits16` array in your program code, it is usually easier to create a black-and-white cursor in a cursor resource, which is described on page 8-33.

A cursor is a 256-pixel, black-and-white image in a 16-by-16 pixel square usually defined by an application in a cursor (`'CURS'`) resource. When your application uses the `GetCursor` function (described on page 8-24) to get a cursor from a `'CURS'` resource, `GetCursor` uses the Resource Manager to load the resource into memory as a `Cursor` record. Your application can then display the color cursor by using the `SetCursor` procedure, which is described on page 8-25.

A `Cursor` record is defined as follows:

```
TYPE CursPtr   = ^Cursor;
CursHandle     = ^CursPtr;
Cursor =
   RECORD
      data:    Bits16;  {cursor image}
      mask:    Bits16;  {cursor mask}
      hotSpot: Point;   {point aligned with mouse}
   END;
```

Field descriptions

`data` Cursor image data, which must begin on a word boundary. The `Bits16` data type for this field is described in the preceding section.

`mask` The cursor's mask, whose effects are shown in Table 8-1. QuickDraw uses the mask to crop the cursor's outline into a background color or pattern. QuickDraw then draws the cursor into this shape. The `Bits16` data type for this field is described in the preceding section.

`hotSpot` A point in the image that aligns with the mouse location.This field aligns a point (not a bit) in the image with the mouse location on the screen. Whenever the user moves the mouse, the low-level interrupt-driven mouse routines move the cursor. When the user clicks, the Event Manager function `WaitNextEvent` reports the location of the cursor's hot spot in global coordinates.

The cursor appears on the screen as a 16-by-16 pixel square. The appearance of each bit of the square is determined by the corresponding bits in the data and mask and, if the mask bit is 0, by the pixel under the cursor, as shown in Table 8-1.

Table 8-1 Cursor appearance

Data	Mask	Resulting pixel on screen
0	1	White
1	1	Black
0	0	Same as pixel under cursor
1	0	Inverse of pixel under cursor

Notice that if all mask bits are 0, the cursor is completely transparent, in that the image under the cursor can still be viewed. Pixels under the white part of the cursor appear unchanged; under the black part of the cursor, black pixels show through as white.

Basic QuickDraw supplies a predefined cursor in the global variable named `arrow`; this is the standard arrow cursor.

CCrsr

Your application typically does not create `CCrsr` records, which are data structures of type `CCrsr`. Although you can create a `CCrsr` record, it is usually easier to create a color cursor in a color cursor resource, which is described on page 8-34.

A color cursor is a 256-pixel color image in a 16-by-16 pixel square usually defined in a color cursor (`'crsr'`) resource. When your application uses the `GetCCursor` function (described on page 8-26) to get a color cursor from a `'crsr'` resource, `GetCCursor` uses the Resource Manager to load the resource into memory as a `CCrsr` record. Your application can then display the color cursor by using the `SetCCursor` procedure, which is described on page 8-26.

The `CCrsr` record is substantially different from the `Cursor` record described in the preceding section; the fields `crsr1Data`, `crsrMask`, and `crsrHotSpot` in the `CCrsr` record are the only ones that have counterparts in the `Cursor` record. A `CCrsr` record is defined as follows:

```
TYPE CCrsrHandle =   ^CCrsrPtr;
CCrsrPtr =           ^CCrsr;
CCrsr =
   RECORD
      crsrType:      Integer;       {type of cursor}
      crsrMap:       PixMapHandle;  {the cursor's PixMap record}
      crsrData:      Handle;        {cursor's data}
      crsrXData:     Handle;        {expanded cursor data}
      crsrXValid:    Integer;       {depth of expanded data}
      crsrXHandle:   Handle;        {reserved for future use}
      crsr1Data:     Bits16;        {1-bit cursor}
      crsrMask:      Bits16;        {cursor's mask}
      crsrHotSpot:   Point;         {cursor's hot spot}
      crsrXTable:    LongInt;       {private}
      crsrID:        LongInt;       {ctSeed for expanded cursor}
   END;
```

Field descriptions

`crsrType`	The type of cursor. Possible values are $8000 for a black-and-white cursor and $8001 for a color cursor.
`crsrMap`	A handle to the `PixMap` record defining the cursor's characteristics. `PixMap` records are described in the chapter "Color QuickDraw" in this book.
`crsrData`	A handle to the cursor's pixel data.
`crsrXData`	A handle to the expanded pixel image used internally by Color QuickDraw.
`crsrXValid`	The depth of the expanded cursor image. If you change the cursor's data or color table, you should set this field to 0 to cause the cursor to be re-expanded. You should never set it to any other values.
`crsrXHandle`	Reserved for future use.
`crsr1Data`	A 16-by-16 pixel image with a pixel depth of 1 to be displayed when the cursor is on screens with pixel depths of 1 or 2 bits.
`crsrMask`	The cursor's mask data. QuickDraw uses the mask to crop the cursor's outline into a background color or pattern. QuickDraw then draws the cursor into this shape. The same 1-bit mask is used with images specified by the `crsrData` and `crsr1Data` fields.
`crsrHotSpot`	The cursor's hot spot.
`crsrXTable`	Reserved for future use.
`crsrID`	The color table seed for the cursor.

The first four fields of the `CCrsr` record are similar to the first four fields of the `PixPat` record, and are used in the same manner by Color QuickDraw. See the chapter "Color QuickDraw" in this book for information about `PixPat` records.

The display of a cursor involves a relationship between a mask, stored in the `crsrMask` field with the same format used for 1-bit cursor masks, and an image. There are two possible sources for a color cursor's image. When the cursor is on a screen whose depth is 1 or 2 bits per pixel, the image for the cursor is taken from the `crsr1Data` field, which contains bitmap cursor data (similar to the bitmap in a `'CURS'` resource).

When the screen depth is greater than 2 bits per pixel, the `crsrMap` field and the `crsrData` field define the image. The pixels within the mask replace the destination pixels. Color QuickDraw transfers the pixels outside the mask into the destination pixels using the XOR Boolean transfer mode. Therefore, if pixels outside the mask are white, the destination pixels aren't changed. If pixels outside the mask are all black, the destination pixels are inverted. All other values outside of the mask cause unpredictable results. See the discussion of Boolean transfer modes in the chapter "Color QuickDraw" in this book for more information about the XOR Boolean transfer mode.

To work properly, a color cursor's image should contain white pixels (R = G = B = $FFFF) for the transparent part of the image, and black pixels (R = G = B = $0000) for the part of the image to be inverted, in addition to the other colors in the cursor's image. Thus, to define a cursor that contains two colors, it's necessary to use a 2-bit cursor image (that is, a four-color image).

If your application changes the value of your color cursor data or its color table, it should set the `crsrXValid` field to 0 to indicate that the color cursor's data needs to be re-expanded, and it should assign a new unique value to the `crsrID` field (unique values can be obtained using the Color Manager function `GetCTSeed`, which is described in *Inside Macintosh: Advanced Color Imaging*. Then your application should call `SetCCursor` to display the changed color cursor.

Cursors

When passing a value to the `Show_Cursor` procedure (described on page 8-30), you can use the `Cursors` data type to represent the kind of cursor to show. The `Cursors` data type is defined as follows:

```
TYPE Cursors =      {values to pass to Show_Cursor}
  (HIDDEN_CURSOR,   {the current cursor}
   I_BEAM_CURSOR,   {the I-beam cursor; to select text}
   CROSS_CURSOR,    {the crosshairs cursor; to draw }
                    { graphics}
   PLUS_CURSOR,     {the plus sign cursor; to select }
                    { cells}
   WATCH_CURSOR,    {the wristwatch cursor; to }
                    { indicate a short operation in }
                    { progress}
   ARROW_CURSOR);   {the standard cursor}
```

Acur

Your application typically does not create `Acur` records, which are data structures of type `Acur`. Although you can create an `Acur` record, which specifies the `'CURS'` resources to use in an animated cursor sequence, it is usually easier to create an animated cursor (`'acur'`) resource, which is described on page 8-36.

When your application uses the `InitCursorCtl` procedure (described on page 8-22), the Resource Manager loads an animated cursor resource into memory as an `Acur` record, which in turn is used by the `RotateCursor` procedure or `SpinCursor` procedure (both described on page 8-32) when sequencing through `'CURS'` resources.

An `Acur` resource is defined as follows:

```
TYPE acurPtr = ^Acur;
acurHandle = ^acurPtr;
Acur =
   RECORD
      n:       Integer;    {number of cursors ("frames")}
      index:   Integer;    {reserved}
      frame1:  Integer;    {'CURS' resource ID for frame #1}
      fill1:   Integer;    {reserved}
      frame2:  Integer;    {'CURS' resource ID for frame #2}
      fill2:   Integer;    {reserved}
      frameN:  Integer;    {'CURS' resource ID for frame #N}
      fillN:   Integer;    {reserved}
   END;
```

Field descriptions

`n`	The number of frames in the animated cursor.
`index`	Used by basic QuickDraw to create the animation.
`frame1`	The resource ID of the cursor (`'CURS'`) resource for the first frame sequence of the animation. The cursor resource is described on page 8-33.
`fill1`	Reserved.
`frame2`	The resource ID of the cursor resource for the next frame in the sequence of the animation.
`fill2`	Reserved.
`frameN`	The resource ID of the cursor resource for the last frame used in the sequence of the animation.
`fillN`	Reserved.

Routines

This section describes the routines you use to initialize the cursor, manage a black-and-white cursor, manage a color cursor, hide and show the cursor, and display an animated cursor.

Initializing Cursors

When your application starts up, the Finder sets the cursor to a wristwatch; this indicates that a short operation is in progress. When your application nears completion of its initialization tasks, it should call the `InitCursor` procedure to change the cursor from a wristwatch to an arrow.

If your application uses an animated cursor to indicate that an operation of medium length is under way, it should also call the `InitCursorCtl` procedure to load its `'acur'` resource and associated `'CURS'` resources.

InitCursor

You use the `InitCursor` procedure to set the current cursor to the standard arrow and make it visible.

```
PROCEDURE InitCursor;
```

DESCRIPTION

The `InitCursor` procedure sets the current cursor to the standard arrow and sets the cursor level to 0, making the cursor visible. (A value of –1 makes the cursor invisible.) The cursor level keeps track of the number of times the cursor has been hidden to compensate for nested calls to the `HideCursor` and `ShowCursor` procedures.

SEE ALSO

For a description of the `HideCursor` procedure, see page 8-28. For a description of the `ShowCursor` procedure, see page 8-30. Listing 8-1 on page 8-6 illustrates how to use the `InitCursor` procedure.

InitCursorCtl

To load the resources necessary for displaying an animated cursor, use the `InitCursorCtl` procedure.

```
PROCEDURE InitCursorCtl (newCursors: UNIV acurHandle);
```

`newCursors`

A handle to an `Acur` record (described on page 8-20) that specifies the cursor resources you want to use in your animation. If you specify `NIL` in this parameter, `InitCursorCtl` loads the animated cursor resource (described on page 8-36) with resource ID 0—as well as the cursor resources (described on page 8-33) specified therein—from your application's resource file.

DESCRIPTION

The `InitCursorCtl` procedure loads the cursor resources for an animated cursor sequence into memory. Your application should call the `InitCursorCtl` procedure once prior to calling the `RotateCursor` procedure (described on page 8-32) or the `SpinCursor` procedure (described on page 8-32).

If your application passes `NIL` in the `newCursors` parameter, `InitCursorCtl` loads the `'acur'` resource with resource ID 0, as well as the `'CURS'` resources whose resource IDs are specified in the `'acur'` resource. If any of the resources cannot be loaded, the cursor does not change when you call `RotateCursor` or `SpinCursor`. Otherwise, the `RotateCursor` procedure and the `SpinCursor` procedure display in sequence the cursors specified in these resources.

If your application does not pass `NIL` in the `newCursors` parameter, it must pass a handle to an `Acur` record. Your application can use the Resource Manager function `GetResource` to obtain a handle to an `'acur'` resource, which your application should then coerce to a handle of type `acurHandle` when passing it to `InitCursorCtl`.

If your application calls the `RotateCursor` or `SpinCursor` procedure without calling `InitCursorCtl`, `RotateCursor` and `SpinCursor` automatically call `InitCursorCtl`. However, since you won't know the state of memory, any memory allocated by the Resource Manager for animating cursors may load into an undesirable location, possibly causing fragmentation. Calling the `InitCursorCtl` procedure during your initialization process has the advantage of causing the memory allocation when you can control its location. For information on using the `InitCursorCtl` procedure during your initialization process, see "Initializing the Cursor" on page 8-6.

SPECIAL CONSIDERATION

If you want to use multiple `'acur'` resources repeatedly during the execution of your application, be aware that the `InitCursorCtl` procedure changes each `frameN` and `fillN` integer pair within the `Acur` record in memory to a handle to the corresponding `'CURS'` resource, which is also in memory. Thus, if the `newCursors` parameter is not `NIL` when your application calls the `InitCursorCtl` procedure, your application must guarantee that `newCursors` always points to a fresh copy of an `'acur'` resource.

SEE ALSO

Listing 8-1 on page 8-6 illustrates how to initialize an animated cursor by using the `InitCursorCtl` procedure. Listing 8-3 on page 8-15 shows how to animate the cursor with the `RotateCursor` procedure, and Listing 8-4 on page 8-15 shows how to animate the cursor with the `SpinCursor` procedure.

Changing Black-and-White Cursors

When you use the `InitCursor` procedure described on page 8-22, the cursor changes from a wristwatch to an arrow. You can change the cursor to another shape by using the `GetCursor` function to load another cursor into memory and then using the `SetCursor` procedure to display it on the screen.

GetCursor

You use the `GetCursor` function to load a cursor resource (described on page 8-33) into memory. You can then display the cursor specified in this resource by calling the `SetCursor` procedure (described in the next section).

```
FUNCTION GetCursor (cursorID: Integer): CursHandle;
```

`cursorID` The resource ID for the cursor you want to display. You can supply one of these constants to get a handle to one of the standard cursors:

```
CONST
   iBeamCursor = 1;  {to select text}
   crossCursor = 2;  {to draw graphics}
   plusCursor  = 3;  {to select cells}
   watchCursor = 4;  {to indicate a short operation }
                     { in progress}
```

DESCRIPTION

The `GetCursor` function returns a handle to a `Cursor` record (described on page 8-16) for the cursor with the resource ID that you specify in the `cursorID` parameter. If the resource can't be read into memory, `GetCursor` returns `NIL`.

To get a handle to a color cursor, use the `GetCCursor` function, which is described on page 8-26.

SEE ALSO

Listing 8-2 on page 8-10 illustrates how to use the `GetCursor` and `SetCursor` routines to change the cursor's shape.

SetCursor

After using the `GetCursor` function to return a handle to a cursor as described in the preceding section, you can use the `SetCursor` procedure to make that cursor the current cursor.

```
PROCEDURE SetCursor (crsr: Cursor);
```

`crsr` A `Cursor` record, as described on page 8-16.

DESCRIPTION

The `SetCursor` procedure displays the cursor you specify in the `crsr` parameter. If the cursor is hidden, it remains hidden and attain its new appearance only when it's uncovered. If the cursor is already visible, it changes to the new appearance immediately.

You need to use the `InitCursor` procedure (described on page 8-22) to initialize the standard arrow cursor and make it visible on the screen before you can call `SetCursor` to change the cursor's appearance.

To display a color cursor, you must use the `SetCCursor` procedure, which is described on page 8-26.

SEE ALSO

Listing 8-2 on page 8-10 illustrates how to use the `GetCursor` and `SetCursor` routines to change the cursor's shape.

Changing Color Cursors

This section describes how to create and display color cursors on the screen. It might be useful to display a color cursor when the user is drawing or typing in color. For example, the insertion point could appear in the color that is being used. Except for multicolored paintbrush cursors, the cursor shouldn't contain more than one color at once because it's hard for the eye to distinguish small areas of color.

To display a color cursor, you load the cursor resource into memory using the `GetCCursor` function. Then you specify the cursor to display on the screen using the `SetCCursor` procedure. Use the `DisposeCCursor` procedure to release the memory used by the color cursor. Although you should never need to do so (because Color QuickDraw handles this), the `AllocCursor` procedure reallocates cursor memory.

GetCCursor

You use the `GetCCursor` function to load a color cursor resource into memory.

```
FUNCTION GetCCursor (crsrID: Integer): CCrsrHandle;
```

`crsrID` The resource ID of the cursor that you want to display.

DESCRIPTION

The `GetCCursor` function creates a new `CCrsr` record and initializes it using the information in the `'crsr'` resource with the specified ID. The `GetCCursor` function returns a handle to the new `CCrsr` record. You can then display this cursor on the screen by calling `SetCCursor`. If a resource with the specified ID isn't found, then this function returns a `NIL` handle.

Since the `GetCCursor` function creates a new `CCrsr` record each time it is called, your application shouldn't call the `GetCCursor` function before each call to the `SetCCursor` procedure (unlike the way `GetCursor` and `SetCursor` are normally used). The `GetCCursor` function doesn't dispose of or detach the resource, so resources of type `'crsr'` should typically be purgeable. You should call the `DisposeCCursor` procedure (described on page 8-27) when you are finished using the color cursor created with `GetCCursor`.

SEE ALSO

For a description of the `'crsr'` resource format, see page 8-34. For a description of the `CCrsr` record, see page 8-18. For a description of the `SetCCursor` procedure, see the next section.

SetCCursor

You use the `SetCCursor` procedure to specify a color cursor for display on the screen.

```
PROCEDURE  SetCCursor (cCrsr: CCrsrHandle);
```

`cCrsr` A handle to the color cursor to be displayed.

DESCRIPTION

The `SetCCursor` procedure allows your application to set a color cursor for display on the screen. At the time the cursor is set, it's expanded to the current screen depth so that it can be drawn rapidly. You must call `GetCCursor` before you call `SetCCursor`; however, you can make several subsequent calls to `SetCCursor` once `GetCCursor` creates the `CCrsr` record.

If your application has changed the cursor's data or its color table, it must also invalidate the `crsrXValid` and `crsrID` fields of the `CCrsr` record before calling `SetCCursor`.

DisposeCCursor

You use the `DisposeCCursor` procedure to dispose of all records allocated by the `GetCCursor` function. The `DisposeCCursor` procedure is also available as the `DisposCCursor` procedure.

```
PROCEDURE DisposeCCursor (cCrsr: CCrsrHandle);
```

`cCrsr` A handle to the color cursor to be disposed of.

DESCRIPTION

The `DisposeCCursor` procedure disposes of memory allocated by the `GetCCursor` function. You should use `DisposeCCursor` for each call to the `GetCCursor` function (described on page 8-26).

AllocCursor

Although you typically won't need to, you can use the `AllocCursor` procedure to reallocate cursor memory.

```
PROCEDURE AllocCursor;
```

DESCRIPTION

Under normal circumstances, you should never need to use this procedure, since Color QuickDraw handles reallocation of cursor memory.

Hiding and Showing Cursors

You can remove the cursor image from the screen by using either the `HideCursor` or `Hide_Cursor` procedure. You can hide the cursor temporarily by using the `ObscureCursor` procedure, or you can hide the cursor in a given rectangle by using the `ShieldCursor` procedure. Your application should hide the cursor when the user is typing, for example. To display a cursor hidden by the `HideCursor`, `Hide_Cursor`, or `ObscureCursor` procedure, use the `ShowCursor` or `Show_Cursor` procedure. (When you use `ObscureCursor` to hide the cursor, the cursor is redisplayed automatically the next time the user moves the mouse.)

HideCursor

You can use the `HideCursor` procedure to remove the cursor from the screen.

```
PROCEDURE HideCursor;
```

DESCRIPTION

The `HideCursor` procedure removes the cursor from the screen, restores the bits under the cursor image, and decrements the cursor level (which `InitCursor` initialized to 0). You might want to use `HideCursor` when the user is using the keyboard to create content in one of your application's windows. Every call to `HideCursor` should be balanced by a subsequent call to the `ShowCursor` procedure, which is described on page 8-30.

Hide_Cursor

You can use the `Hide_Cursor` procedure to hide the cursor if it is visible on the screen. The `Hide_Cursor` procedure is functionally the same as the `HideCursor` procedure described in the preceding section.

```
PROCEDURE Hide_Cursor;
```

DESCRIPTION

The `Hide_Cursor` procedure calls the `HideCursor` procedure to remove the cursor's image from the screen and decrements the cursor level by 1. Every call to `Hide_Cursor` should be balanced by a subsequent call to the `Show_Cursor` procedure, which is described on page 8-30. Before using `Hide_Cursor`, you must use the `InitCursorCtl` procedure, which is described on page 8-22.

ObscureCursor

You use the `ObscureCursor` procedure to hide the cursor until the next time the user moves the mouse.

```
PROCEDURE ObscureCursor;
```

DESCRIPTION

The `ObscureCursor` procedure temporarily hides the cursor; the cursor is redisplayed the next time the user moves the mouse. Your application normally calls `ObscureCursor` when the user begins to type. Unlike `HideCursor` (which is described on page 8-28), `ObscureCursor` has no effect on the cursor level and must not be balanced by a call to `ShowCursor`.

ShieldCursor

You can use the `ShieldCursor` procedure to hide the cursor in a rectangle.

```
PROCEDURE ShieldCursor (shieldRect: Rect; offsetPt: Point);
```

`shieldRect`
A rectangle in which the cursor is hidden whenever the cursor intersects the rectangle. The rectangle may be specified in global or local coordinates. If you are using global coordinates, pass (0,0) in the `offsetPt` parameter. If you are using the local coordinates of a graphics port, pass the coordinates for the upper-left corner of the graphics port's boundary rectangle in the `offsetPt` parameter.

`offsetPt` A point value for the offset of the rectangle. Like the basic QuickDraw procedure `LocalToGlobal`, the `ShieldCursor` procedure offsets the coordinates of the rectangle by the coordinates of this point.

DESCRIPTION

If the cursor and the given rectangle intersect, `ShieldCursor` hides the cursor. If they don't intersect, the cursor remains visible while the mouse isn't moving, but is hidden when the mouse moves. This procedure may be useful when using a feature such as QuickTime to display content in a specified rectangle. When a QuickTime movie is animating, the cursor should not be visible in front of the movie.

The `ShieldCursor` procedure decrements the cursor level and should be balanced by a call to the `ShowCursor` procedure, which is described in the next section.

ShowCursor

You use the `ShowCursor` procedure to display a cursor hidden by the `HideCursor` or `ShieldCursor` procedure.

```
PROCEDURE ShowCursor;
```

DESCRIPTION

The `ShowCursor` procedure increments the cursor level, which may have been decremented by the `HideCursor` or `ShieldCursor` procedure, and displays the cursor on the screen when the level is 0. A call to the `ShowCursor` procedure should balance each previous call to the `HideCursor` or `ShieldCursor` procedure. The level isn't incremented beyond 0, so extra calls to `ShowCursor` have no effect.

Low-level interrupt-driven routines link the cursor with the mouse position, so that if the cursor level is 0 (visible), the cursor automatically follows the mouse.

If the cursor has been changed with the `SetCursor` procedure while hidden, `ShowCursor` displays the new cursor.

SEE ALSO

For a description of the `HideCursor` procedure, see page 8-28. The `ShieldCursor` procedure is described on page 8-29, and the `SetCursor` procedure is described on page 8-25.

Show_Cursor

You use the `Show_Cursor` procedure to display the cursor on the screen if you have used the `Hide_Cursor` procedure (described on page 8-28) to remove the cursor from the screen.

```
PROCEDURE Show_Cursor (cursorKind: Cursors);
```

`cursorKind`

The kind of cursor to show. To specify one of the standard cursors, you can use one of these values defined by the `Cursors` data type.

```
TYPE Cursors =      {values to pass Show_Cursor}
  (HIDDEN_CURSOR,   {the current cursor}
   I_BEAM_CURSOR,   {the I-beam cursor; to select text}
   CROSS_CURSOR,    {the crosshairs cursor; to draw }
                    { graphics}
   PLUS_CURSOR,     {the plus sign cursor; to select }
                    { cells}
```

```
WATCH_CURSOR,    {the wristwatch cursor; to }
                 { indicate a short operation in }
                 { progress}
ARROW_CURSOR);   {the standard cursor}
```

DESCRIPTION

The `Show_Cursor` procedure increments the cursor level, which may have been decremented by the `Hide_Cursor` procedure, and displays the specified cursor on the screen only if the level becomes 0 (it is never incremented beyond 0). You can specify one of the standard cursors or the current cursor by passing one of the previously listed values in the `cursorKind` parameter. If you specify one of the standard cursors, the `Show_Cursor` procedure calls the `SetCursor` procedure for the specified cursor prior to calling `ShowCursor`. If you specify `HIDDEN_CURSOR`, this procedure just calls `ShowCursor`. Before using `Show_Cursor`, you must use the `InitCursorCtl` procedure, which is described on page 8-22.

SPECIAL CONSIDERATIONS

The value `ARROW_CURSOR` works correctly only if the basic QuickDraw global variables have been set up by using the `InitGraf` procedure, which is described in the chapter "Basic QuickDraw" in this book.

SEE ALSO

Figure 8-3 on page 8-8 illustrates the cursors represented by the `Cursors` data type.

Displaying Animated Cursors

This section describes how to display an animated cursor using the `RotateCursor` procedure or the `SpinCursor` procedure. You use an animated cursor when your application performs a medium-length operation that might cause the user to think that the computer has quit working. The two procedures are similar, but you must maintain a counter with the `RotateCursor` procedure.

You need to call the `InitCursorCtl` procedure to load your cursor resources before using the routines described in this section. For information about using the `InitCursorCtl` procedure, see page 8-22.

RotateCursor

You can use the `RotateCursor` procedure to display an animated cursor when your application performs a medium-length operation that might cause the user to think that the computer has quit working.

```
PROCEDURE RotateCursor (counter: LongInt);
```

`counter` An incrementing or decrementing index maintained by your application. When the index is a multiple of 32, the next cursor frame is used in the animation. A positive counter moves forward through the cursor frames, and a negative counter moves backward through the cursor frames.

DESCRIPTION

The `RotateCursor` procedure animates whatever sequence of cursors you set up by using the `InitCursorCtl` procedure. If the value of `counter` is a multiple of 32, the `RotateCursor` procedure calls the `SetCursor` procedure to set the cursor to the next cursor frame. `RotateCursor` does not show the cursor if it is currently hidden. If the cursor is hidden, you can show it by making a call to `ShowCursor` or `Show_Cursor` (both described on page 8-30).

SEE ALSO

For an example of using the `RotateCursor` procedure, see Listing 8-3 on page 8-15.

SpinCursor

You can use the `SpinCursor` procedure to display an animated cursor when your application performs a medium-length operation that might cause the user to think that the computer has quit working.

```
PROCEDURE SpinCursor (increment: Integer);
```

`increment` A value that determines the sequencing direction of the cursor. A positive increment moves forward through the cursor frames, and a negative increment moves backward through the cursor frames. A 0 value for the increment resets the counter to 0 and steps to the next cursor frame.

DESCRIPTION

The `SpinCursor` procedure is similar to the `RotateCursor` procedure except that, instead of passing a counter, you pass a value that indicates which direction to spin the cursor. Your application is responsible for determining the proper intervals at which to call `SpinCursor`. Your application specifies the increment to be counted, either positive or negative, and `SpinCursor` adds the increment to its counter. The sign of the increment, not the sign of the accumulated value of the `SpinCursor` counter, determines the cursor's direction of spin.

SEE ALSO

For an example of using the `SpinCursor` procedure, see Listing 8-4 on page 8-15.

Resources

This section describes the cursor (`'CURS'`) resource, the color cursor (`'crsr'`) resource, and the animated cursor (`'acur'`) resource. Your application can use a `'CURS'` resource to create a black-and-white cursor other than the standard cursors or a `'crsr'` resource to create a color cursor to display on color screens. Your application can use an `'acur'` resource to create an animated cursor to display when a medium-length operation is taking place. These resource types should be marked as purgeable. See the discussion of the pointing device in *Macintosh Human Interface Guidelines* for more information on when to use different types of cursors in your application; see also the discussion of color in the same book.

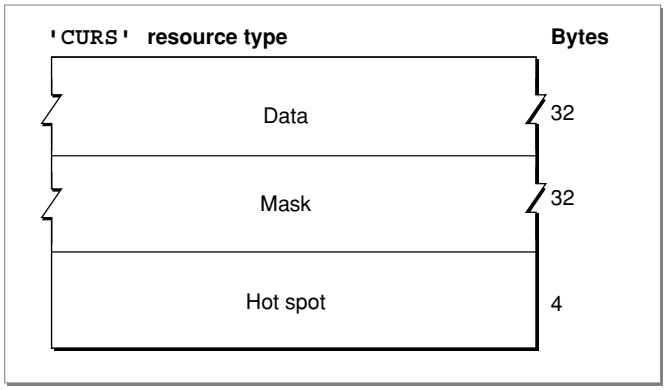
The Cursor Resource

You can use a cursor resource to define a cursor to display in your application. A cursor resource is a resource of type `'CURS'`. All cursor resources must be marked purgeable and must have resource IDs greater than 128. You use the `GetCursor` function (described on page 8-24) to obtain a cursor stored in a `'CURS'` resource. QuickDraw reads the requested resource, copies it, and then alters the copy before passing it to your application.

This section describes the structure of this resource after it has been compiled by the Rez resource compiler, available from APDA. However, you typically use a high-level utility such as the ResEdit application to create `'CURS'` resources. You can then use the DeRez decompiler to convert your `'CURS'` resources into Rez input when necessary.

The compiled output format for a 'CURS' resource is illustrated in Figure 8-8.

Figure 8-8 Format of a compiled cursor ('CURS') resource



The compiled version of a 'CURS' resource contains the following elements:

- Data. A bitmap for the cursor.
- Mask. A bitmap for the cursor's mask. QuickDraw uses the mask to crop the cursor's outline into a background color or pattern. QuickDraw then draws the cursor into this shape.
- Hot spot. The cursor's hot spot.

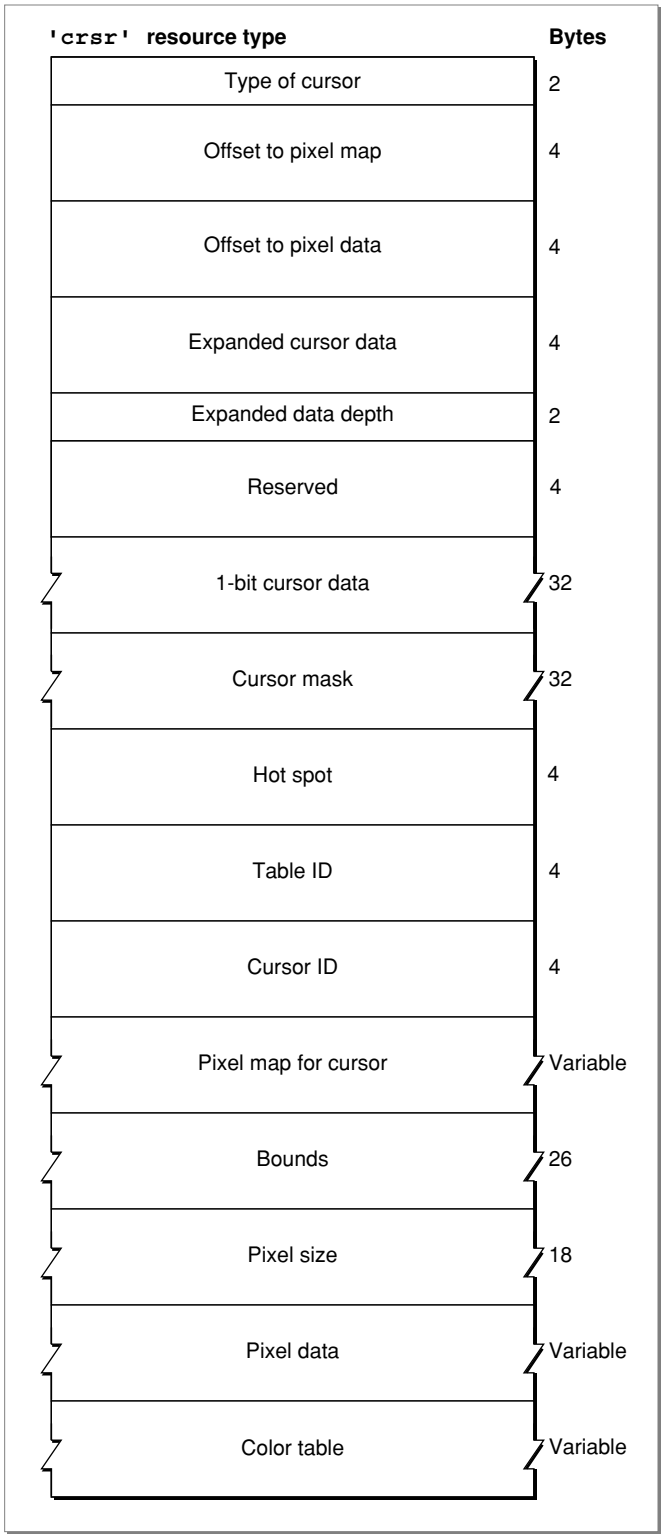
The Color Cursor Resource

You can use a color cursor resource to define a colored cursor to display in your application. A color cursor resource is a resource of type 'crsr'. All color cursor resources must be marked purgeable and must have resource IDs greater than 128. You use the GetCCursor function (described on page 8-26) to obtain a color cursor stored in a 'crsr' resource. Color QuickDraw reads the requested resource, copies it, and then alters the copy before passing it to the application. Each time you call GetCCursor, you get a new copy of the cursor. This means that you should call GetCCursor only once for a color cursor, even if you call the SetCCursor procedure many times.

This section describes the structure of this resource after it has been compiled by the Rez resource compiler, available from APDA. However, you typically use a high-level utility such as the ResEdit application to create 'crsr' resources. You can then use the DeRez decompiler to convert your 'crsr' resources into Rez input when necessary.

The compiled output format for a 'crsr' resource is illustrated in Figure 8-9.

Figure 8-9 Format of a compiled color cursor (`'crsr'`) resource



The compiled version of a `'crsr'` resource contains the following elements:

- Type of cursor. A value of $8001 identifies this as a color cursor. A value of $8000 identifies this as a black-and-white cursor.
- Offset to `PixMap` record. This offset is from the beginning of the resource data.
- Offset to pixel data. This offset is from the beginning of the resource data.
- Expanded cursor data. This expanded pixel image is used internally by Color QuickDraw.
- Expanded data depth. This is the pixel depth of the expanded cursor image.
- Reserved. The Resource Manager uses this element for storage.
- Cursor data. This field contains a 16-by-16 pixel 1-bit image to be displayed when the cursor is on 1-bit or 2-bit screens.
- Cursor mask. A bitmap for the cursor's mask. QuickDraw uses the mask to crop the cursor's outline into a background color or pattern. QuickDraw then draws the cursor into this shape.
- Hot spot. The cursor's hot spot.
- Table ID. This contains an offset to the color table data from the beginning of the resource data.
- Cursor ID. This contains the cursor's resource ID.
- Pixel map. This pixel map describes the image when drawing the color cursor. The pixel map contains an offset to the color table data from the beginning of the resource.
- Bounds. The boundary rectangle of the cursor.
- Pixel size. The number of pixels per bit in the cursor.
- Pixel data. The data for the cursor.
- Color table. A color table containing the color information for the cursor's pixel map.

The Animated Cursor Resource

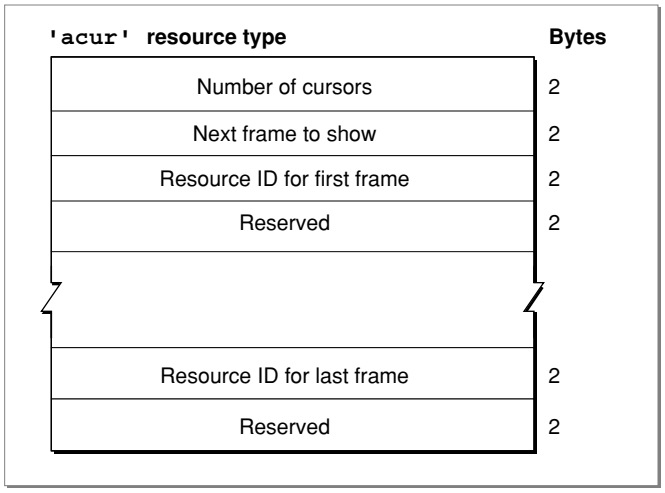
You can use an animated cursor resource to define a set of frames for an animated cursor to display in your application. An animated cursor resource is a resource of type `'acur'`.

If you pass `NIL` to `InitCursorCtl` (described on page 8-22), it automatically loads the `'acur'` resource that has an ID of 0 in your application's resource file. If you wish to use multiple `'acur'` resources, you must give them resources IDs greater than 128, and you must use the Resource Manager function `GetResource` to obtain handles to them. You must then coerce their handles to type `acurHandle`, which you pass to `InitCursorCtl`. You use the `SpinCursor` or `RotateCursor` procedure to animate the cursors stored in an `'acur'` resource.

This section describes the structure of this resource after it has been compiled by the Rez resource compiler, available from APDA. However, you typically use a high-level tool such as the ResEdit application to create `'acur'` resources. You can then use the DeRez decompiler to convert your `'acur'` resources into Rez input when necessary.

The compiled output format for an `'acur'` resource is illustrated in Figure 8-10.

Figure 8-10 Format of a compiled animated cursor (`'acur'`) resource



The compiled version of an `'acur'` resource contains the following elements:

- Number of cursors. The number of frames used to animate the cursor.
- Next frame to show. Reserved.
- Resource ID of the cursor resource that defines the first frame of the animation.
- Reserved.
- Resource ID of the cursor resource that defines the last frame of the animation.
- Reserved.

Summary of Cursor Utilities

Pascal Summary

Constants

```
CONST
   iBeamCursor = 1;  {used in text editing}
   crossCursor = 2;  {often used for manipulating graphics}
   plusCursor  = 3;  {often used for selecting fields in an array}
   watchCursor = 4;  {used to mean a short operation is in progress}
```

Data Types

```
TYPE  Bits16 = ARRAY[0..15] OF Integer;

      CursPtr = ^Cursor;
      CursHandle = ^CursPtr;
      Cursor =
      RECORD
         data:    Bits16;  {cursor image}
         mask:    Bits16;  {cursor mask}
         hotSpot: Point;   {point aligned with mouse}
      END;

      CCrsrPtr = ^CCrsr;
      CCrsrHandle = ^CCrsrPtr;
      CCrsr =
      RECORD
         crsrType:   Integer;       {type of cursor}
         crsrMap:    PixMapHandle;  {the cursor's PixMap record}
         crsrData:   Handle;        {cursor's data}
         crsrXData:  Handle;        {expanded cursor data}
         crsrXValid: Integer;       {depth of expanded data (0 if none)}
         crsrXHandle:Handle;        {future use}
```

```
   crsr1Data:   Bits16;       {1-bit cursor}
   crsrMask:    Bits16;       {cursor's mask}
   crsrHotSpot: Point;        {cursor's hot spot}
   crsrXTable:  LongInt;      {private}
   crsrID:      LongInt;      {ctSeed for expanded cursor}
END;

Cursors =            {values to pass to Show_Cursor}
   (HIDDEN_CURSOR,   {the current cursor}
    I_BEAM_CURSOR,   {the I-beam cursor; to select text}
    CROSS_CURSOR,    {the crosshairs cursor; to draw graphics}
    PLUS_CURSOR,     {the plus sign cursor; to select cells}
    WATCH_CURSOR,    {the wristwatch cursor; to indicate a }
                     { short operation in progress}
    ARROW_CURSOR);   {the standard cursor}

acurPtr = ^Acur;
acurHandle = ^acurPtr;
Acur =
RECORD
   n:       Integer;    {number of cursors ("frames")}
   index:   Integer;    {reserved}
   frame1:  Integer;    {'CURS' resource ID for frame #1}
   fill1:   Integer;    {reserved}
   frame2:  Integer;    {'CURS' resource ID for frame #2}
   fill2:   Integer;    {reserved}
   frameN:  Integer;    {'CURS' resource ID for frame #N}
   fillN:   Integer;    {reserved}
END;
```

Routines

Initializing Cursors

```
PROCEDURE InitCursor;
PROCEDURE InitCursorCtl     (newCursors: UNIV acurHandle);
```

Changing Black-and-White Cursors

```
FUNCTION GetCursor          (cursorID: Integer): CursHandle;
PROCEDURE SetCursor         (crsr: Cursor);
```

Changing Color Cursors

```
{DisposeCCursor is also spelled as DisposCCursor}
FUNCTION GetCCursor         (cursorID: Integer): CCursHandle;
PROCEDURE SetCCursor        (cCrsr: CCrsrHandle);
PROCEDURE DisposeCCursor    (cCrsr: CCrsrHandle);
PROCEDURE AllocCursor;
```

Hiding and Showing Cursors

```
PROCEDURE HideCursor;
PROCEDURE Hide_Cursor;
PROCEDURE ObscureCursor;
PROCEDURE ShieldCursor      (shieldRect: Rect; offsetPt: Point);
PROCEDURE ShowCursor;
PROCEDURE Show_Cursor       (cursorKind: Cursors);
```

Displaying Animated Cursors

```
PROCEDURE RotateCursor      (counter: LongInt);
PROCEDURE SpinCursor        (increment: Integer);
```

C Summary

Constants

```
enum {
   iBeamCursor =  1,    /* used in text editing */
   crossCursor =  2,    /* often used for manipulating graphics */
   plusCursor =   3,    /* often used for selecting fields in an array */
   watchCursor =  4     /* used to mean a short operation is in progress */
};

enum {               /* values to pass to Show_Cursor */
   HIDDEN_CURSOR,    /* the current cursor */
   I_BEAM_CURSOR,    /* the I-beam cursor; to select tect */
   CROSS_CURSOR,     /* the crosshairs cursor; to draw graphics */
   PLUS_CURSOR,      /* the plus sign cursor; to select cells */
```

```
   WATCH_CURSOR,     /* the wristwatch cursor; to indicate a short
                        operation in progress */
   ARROW_CURSOR      /* the standard cursor */
};
typedef unsigned char Cursors;
```

Data Types

```
typedef short Bits16[16];

struct Cursor {
   Bits16   data;    /* cursor image */
   Bits16   mask;    /* cursor mask */
   Point    hotSpot; /* point aligned with mouse */
};
typedef struct Cursor Cursor;
typedef Cursor *CursPtr, **CursHandle;

struct CCrsr {
   short          crsrType;      /* type of cursor */
   PixMapHandle   crsrMap;       /* the cursor's PixMap record */
   Handle         crsrData;      /* cursor's data */
   Handle         crsrXData;     /* expanded cursor data */
   short          crsrXValid;    /* depth of expanded data (0 if none) */
   Handle         crsrXHandle;   /* future use */
   Bits16         crsr1Data;     /* 1-bit cursor */
   Bits16         crsrMask;      /* cursor's mask */
   Point          crsrHotSpot;   /* cursor's hot spot */
   long           crsrXTable;    /* private */
   long           crsrID;        /* ctSeed for expanded cursor */
};
typedef struct CCrsr CCrsr;
typedef CCrsr *CCrsrPtr, **CCrsrHandle;

struct Acur {
   short    n;       /* number of cursors ("frames of film") */
   short    index;   /* reserved */
   short    frame1;  /* 'CURS' resource ID for frame #1 */
   short    fill1;   /* reserved */
   short    frame2;  /* 'CURS' resource ID for frame #2 */
```

```
   short    fill2;   /* reserved */
   short    frameN;  /* 'CURS' resource ID for frame #N */
   short    fillN;   /* reserved */
};
typedef struct Acur acur,*acurPtr,**acurHandle;
```

Functions

Initializing Cursors

```
pascal void InitCursor      (void);
pascal void InitCursorCtl   (acurHandle newCursors);
```

Changing Black-and-White Cursors

```
pascal CursHandle GetCursor (short cursorID);
pascal void SetCursor       (const Cursor *crsr);
```

Changing Color Cursors

```
/* DisposeCCursor is also spelled as DisposCCursor */
pascal CCrsrHandle GetCCursor
                            (short crsrID);
pascal void SetCCursor      (CCrsrHandle cCrsr);
pascal void DisposeCCursor  (CCrsrHandle cCrsr);
pascal void AllocCursor     (void);
```

Hiding and Showing Cursors

```
pascal void HideCursor      (void);
pascal void Hide_Cursor     (void);
pascal void ObscureCursor   (void);
pascal void ShieldCursor    (const Rect *shieldRect, Point offsetPt);
pascal void ShowCursor      (void);
pascal void Show_Cursor     (Cursors cursorKind);
```

Displaying Animated Cursors

```
pascal void RotateCursor    (long counter);
pascal void SpinCursor      (short increment);
```

Assembly-Language Summary

Data Structures

Cursor Data Structure

0	`data`	32 bytes	cursor image
32	`mask`	32 bytes	cursor mask
64	`hotSpot`	long	point aligned with mouse

Color Cursor Data Structure

0	`crsrType`	word	type of cursor
2	`crsrMap`	long	the cursor's `PixMap` record
6	`crsrData`	long	cursor's data
10	`crsrXData`	long	expanded cursor data
14	`crsrXValid`	word	depth of expanded data (0 if none)
16	`crsrXHandle`	long	handle for future use
20	`crsr1Data`	16 words	1-bit data
52	`crsrMask`	16 words	1-bit mask
84	`crsrHotSpot`	long	hot spot for cursor
88	`crsrXTable`	long	table ID for expanded data
92	`crsrID`	long	ID for cursor
96	`crsrRec`	long	size of cursor save area

Global Variables

`arrow` The standard arrow cursor.

Printing Manager

Contents

About the Printing Manager 9-3
The Printing Graphics Port 9-4
Getting Printing Preferences From the User 9-5
QuickDraw and PostScript Printer Drivers 9-8
Page and Paper Rectangles 9-10
Printer Resolution 9-11
The TPrint Record and the Printing Loop 9-11
Print Status Dialog Boxes 9-13
Using the Printing Manager 9-15
Creating and Using a TPrint Record 9-17
Printing a Document 9-18
Printing From the Finder 9-25
Providing Names of Documents Being Printed 9-27
Printing Hints 9-27
Getting and Setting Printer Information 9-28
Determining and Setting the Resolution of the Current Printer 9-30
Determining Page Orientation 9-32
Enhancing Draft-Quality Printing 9-33
Altering the Style or Job Dialog Box 9-35
Writing an Idle Procedure 9-38
Handling Printing Errors 9-41
Printing Manager Reference 9-43
Data Structures 9-44
Printing Manager Routines 9-57
Opening and Closing the Printing Manager 9-57
Initializing and Validating TPrint Records 9-58
Displaying and Customizing the Print Dialog Boxes 9-61
Printing a Document 9-66
Optimizing Printing 9-72

Handling Printing Errors 9-75
Low-Level Routines 9-78
Application-Defined Routines 9-84
Summary of the Printing Manager 9-87
Pascal Summary 9-87
Constants 9-87
Data Types 9-88
Printing Manager Routines 9-92
Application-Defined Routines 9-93
C Summary 9-94
Constants 9-94
Data Types 9-95
Printing Manager Functions 9-99
Application-Defined Functions 9-101
Assembly-Language Summary 9-101
Data Structures 9-101
Trap Macros 9-103
Global Variable 9-103
Result Codes 9-104

This chapter describes how your application can use the Printing Manager to perform QuickDraw-based printing on a printer connected to a Macintosh computer. The Printing Manager works with the printer driver for the currently selected printer so that your application can draw an image on a printer just as it draws an image on a screen. This allows your application to use the same QuickDraw routines for printing as for screen display.

You should read this chapter if your application allows the user to print. If you want to print with features—such as rotated text and hairlines—that are not supported by QuickDraw, you should read Appendix B, "Using Picture Comments for Printing."

Before reading this chapter, you should be familiar with QuickDraw's drawing routines and the `GrafPort` and `CGrafPort` data types, as described in the chapters "Basic QuickDraw," "QuickDraw Drawing," and "Color QuickDraw" in this book. You may also need to refer to *Inside Macintosh: Text* for information about printing text from non-Roman script systems.

About the Printing Manager

The **Printing Manager** is a collection of system software routines that your application can use to print from the Macintosh computer to any type of connected printer. The Printing Manager is available on all Macintosh computers. When printing, your application calls the same Printing Manager routines regardless of the type of printer selected by the user.

When you print a document using the Printing Manager, the Printing Manager uses a printer driver to do the actual printing. A **printer driver** does any necessary translation of QuickDraw drawing routines and—when requested by your application—sends the translated instructions and data to the printer. Printer drivers are stored in **printer resource files,** which are located in the Extensions folder inside the System Folder. Each type of printer has its own printer driver. One printer driver can communicate with several printers of the same type; for example, the LaserWriter printer driver can work with multiple LaserWriter printers on a network.

The **current printer** is the printer that the user last selected from the Chooser. It is the printer driver for the current printer that actually implements the routines defined by the Printing Manager. Every Printing Manager routine you call determines the current printer from a resource in the System file and then dispatches your call to the printer driver for that printer.

To print a document, your application uses the `PrOpen` procedure to open the driver for the current printer. Your application then uses the `PrOpenDoc` function to open a **printing graphics port,** which is a data structure of type `TPrPort`; it consists of a QuickDraw graphics port (either a `GrafPort` or `CGrafPort` record) plus additional information. For each page in a document, your application uses the `PrOpenPage` procedure to open the page. Your application then uses QuickDraw routines to draw onto the page.

Ideally, your application should be device-independent, so that when it prints a document, it doesn't rely on the presence of any one printer feature. In general, there are two types of printer drivers: those for QuickDraw printers and those for PostScript printers. QuickDraw printer drivers render images using QuickDraw and then send the rendered images to the printer as bitmaps or pixel maps. PostScript printer drivers convert QuickDraw operations into equivalent PostScript operations, as necessary. The driver sends the converted PostScript drawing operations to the printer, which renders the images by interpreting these operations.

For most applications, sending QuickDraw's picture-drawing routines to the printer driver is sufficient: the driver either uses QuickDraw or converts the drawing routines to PostScript. For some applications, such as page-layout programs, this may not be sufficient; such applications may rely on printer drivers to provide several features that are not available, or are difficult to achieve, using QuickDraw.

If your application requires these features (such as rotated text and dashed lines), you may want to create two versions of your drawing code: one that uses picture comments to take advantage of these features on capable printers, and another that provides QuickDraw-based approximations of these features. Created with the QuickDraw procedure `PicComment`, **picture comments** are data or commands used for special processing by output devices, such as printer drivers. Picture comments may be included in the code an application sends to a printer driver, or stored in the definition of a picture. For more information, see Appendix B, "Using Picture Comments for Printing," in this book.

For information about how the PostScript language works and the specifics of PostScript commands, see the *PostScript Language Reference Manual*, second edition, published by Addison-Wesley.

The Printing Graphics Port

You use the `PrOpenDoc` function to open a document for printing. The `PrOpenDoc` function in turn opens a printing graphics port and returns a pointer to a `TPrPort` record, which defines the printing graphics port.

```
TYPE
   TPPrPort =  ^TPrPort;
   TPrPort  =  {printing graphics port record}
   RECORD
      gPort:      GrafPort;   {graphics port for printing}
      gProcs:     QDProcs;    {procedures for printing in the }
                              { graphics port}
      {more fields for internal use}
   END;
```

The graphics port in the `gPort` field is either a `CGrafPort` or `GrafPort` record, depending on whether the current printer supports color and grayscale and whether Color QuickDraw is available on the computer. If you need to determine the type of graphics port, you can check the high bit in the `rowBytes` field of the record contained in the `gPort` field; if this bit is set, the printing graphics port is based on a `CGrafPort` record.

You print text and graphics by drawing into a printing graphics port using QuickDraw drawing routines, just as if you were drawing on the screen. The printer driver installs its own versions of QuickDraw's low-level drawing routines in the `gProcs` field of the `TPrPort` record. Your calls to high-level QuickDraw routines then drive the printer instead of drawing on the screen.

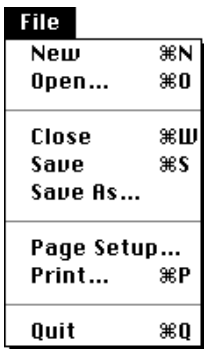
As you draw each page of a document into the printing graphics port, the printer driver translates the calls to QuickDraw routines into the equivalent instructions for the printer. The printer itself does nothing except draw the document on a page, exactly as the printer driver directs it.

Before ever printing a document, however, your application must obtain various printing preferences from the user—usually when the user chooses the Page Setup or Print command from the File menu.

Getting Printing Preferences From the User

If it's likely that a user will want to print the data created with your application, you should support the Page Setup command and the Print command in the File menu. Figure 9-1 shows a typical File menu that includes the Page Setup and Print commands. (See the chapter "Menu Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for detailed information about setting up a File menu with these commands.)

Figure 9-1 A standard File menu for an application



In response to the Page Setup command, your application should display the current printer's **style dialog box,** which allows the user to specify the printing options—such as the paper size and the printing orientation—that your application needs for formatting the document in the frontmost window. In response to the Print command, your application should display the current printer's **job dialog box,** which solicits from the user printing information—such as the number of copies to print, the print quality, and the range of pages to print—for the document in the frontmost window. Each printer driver defines its own style dialog box and job dialog box. Your application can also provide other printing options in these dialog boxes when appropriate.

A `TPrint` record contains the information about the user's choices made with the style and job dialog boxes. When the user saves a document, your application should save the `TPrint` record associated with that document. This allows your application to resume using any style preferences that the user has selected for printing that document. While only the information the user specifies through the style dialog box should be preserved each time the user prints the document, you can save the entire `TPrint` record when you save the document. The information supplied by the user in the job dialog box should pertain to the document only while the document prints; you should not reuse this information if the user prints the document again.

The values that the user specifies through the style dialog box apply only to the printing of the document in the active—that is, frontmost—window. In general, the user should have to specify these preferences only once per document, although the user can choose to change these settings at any time. Figure 9-2 shows the StyleWriter printer driver's style dialog box, displayed by an application in response to the Page Setup command.

Figure 9-2 The style dialog box for a StyleWriter printer

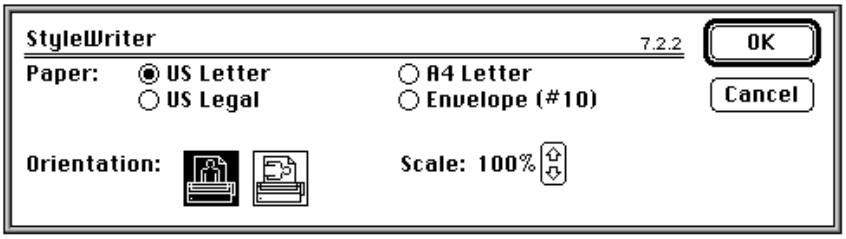
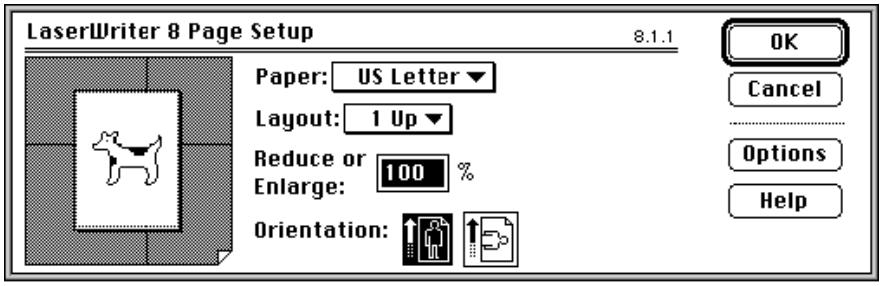


Figure 9-3 shows the style dialog box for a LaserWriter printer. Because each printer resource file defines its own style dialog box, a style dialog box for one printer may differ slightly from that of another printer (as you can see by comparing Figure 9-2 with Figure 9-3).

Figure 9-3 The style dialog box for a LaserWriter printer



You use the `PrStlDialog` function to display the style dialog box defined by the resource file of the current printer. The `PrStlDialog` function handles all user interaction in the items defined by the printer driver until the user clicks the OK or Cancel button. You must call the `PrOpen` procedure prior to calling `PrStlDialog` because the current printer driver must be open for your application to successfully call `PrStlDialog`.

Figure 9-4 shows an example of a job dialog box. Your application should print the document in the active window if the user clicks the Print button.

Figure 9-4 The job dialog box for a StyleWriter printer

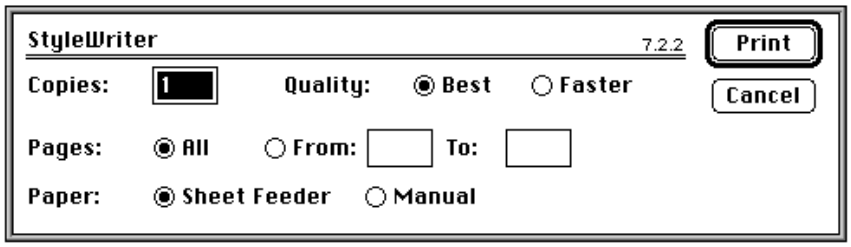
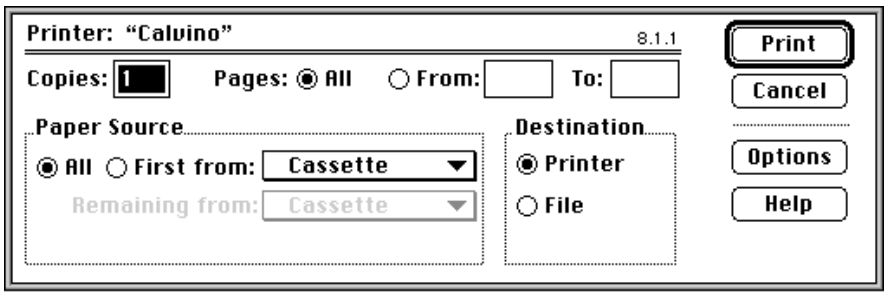


Figure 9-5 shows a job dialog box for a LaserWriter printer.

Figure 9-5 The job dialog box for a LaserWriter printer



Your application uses the `PrJobDialog` function to display the job dialog box defined by the resource file of the current printer. The `PrJobDialog` function handles all user interaction in the items defined by the printer driver until the user clicks the Print or Cancel button.

Your application can customize the style dialog box and the job dialog box to ask for additional information. (Figure 9-12 on page 9-35 shows a print job dialog box that includes two extra checkboxes.) If you customize the style or job dialog box, you must provide a function that handles events, such as clicks, in any items that you add to the dialog box, and you should provide an event filter function to handle events not handled by the Dialog Manager in a modal dialog box.

QuickDraw and PostScript Printer Drivers

There are two main types of printer drivers for Macintosh computers: QuickDraw printer drivers and PostScript printer drivers.

Using QuickDraw drawing operations, **QuickDraw printer drivers** render images on the Macintosh computer and then send the rendered images to the printer in the form of bitmaps or pixel maps. The printer might be a dot-matrix printer, an ink jet printer, a laser printer, or a plotter.

QuickDraw printers are not required to have any intelligent rendering capabilities. Instead, they simply accept instructions from printer drivers to place dots on the page in specific places. A QuickDraw printer driver captures in a temporary disk file (called a **spool file**)—or in memory—the images of an entire page, translates the pixels into dot-placement instructions, and sends these instructions to the printer.

Note
The internal format of spool files is private to the printer drivers and may vary from one printer driver to another. You should not attempt to determine or manipulate the format of these files. ◆

QuickDraw printers are relatively inexpensive to produce because they don't require sophisticated rendering capabilities—instead, they rely on the rendering capabilities of the Macintosh computer. However, QuickDraw printers are also relatively slow. Over 7 million pixels are required to render an 8-by-10-inch image at 300 dpi. Many QuickDraw printers use some form of data compression to improve their performance. For the ImageWriter printer, for example, the printer driver instructs the printer only where to place ink; the printer driver assumes that the rest of the page should remain untouched. Nevertheless, nearly 900 KB is required to render a full-page image at 1 bit per pixel. A color printer that uses 8 bits per pixel requires eight times as much data. Such large memory requirements may require the driver to process the image in horizontal strips, or bands, which further impairs printing speed.

PostScript printer drivers, on the other hand, convert QuickDraw drawing operations into equivalent PostScript drawing operations, as necessary. PostScript printers have their own rendering capabilities. Instead of rendering an entire page on the Macintosh computer and sending all the pixels to the printer, PostScript printer drivers typically send drawing commands directly to the printer, which itself renders images on the page. Many of Apple's LaserWriter printers use the PostScript page-description language to render images in this way, thereby off-loading image processing from the computer. This gives PostScript printers a speed advantage over QuickDraw printers.

QuickDraw printer drivers must capture an entire page before sending any of it to the printer, but PostScript printer drivers may be able to send commands as soon as they are generated to printers. This can result in faster printing, but it doesn't allow the printer driver to examine entire pages for their use of color, fonts, or other resources that the printer needs to have specially processed. Therefore, some PostScript printer drivers may capture page images in a spool file so that the driver can analyze the pages before sending them to the printer.

Some printer drivers allow users to specify **background printing,** which allows the user to work with an application while documents are printing. These printer drivers, which can be either QuickDraw or PostScript, send printing data to a spool file in the PrintMonitor Documents folder inside the System Folder. Do not confuse the different uses of these various spool files. With background printing, print files are spooled to disk to allow the user to work with an application while documents are printing; many printer drivers support background printing regardless of their other capabilities. Some printer drivers create spool files while processing a page image—this, however, does not allow the user to work with the application while the document is printing.

IMPORTANT

There is no reliable manner in which you can determine whether a printer driver creates a spool file—whether for processing of a page image or for background printing. With the exception of using the `PrPicFile` procedure, described on page 9-71, your print loop should not base any of its actions on whether a printer driver creates a spool file. ▲

Page and Paper Rectangles

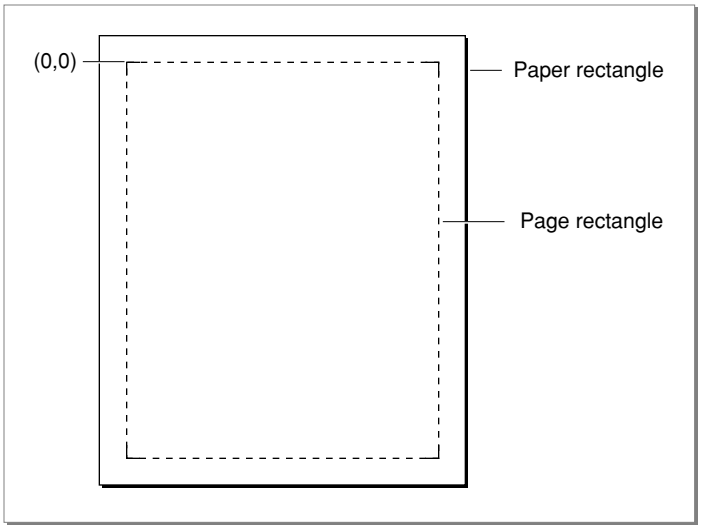
When printing a document, you should consider these two aspects of the layout of the page:

- the physical size of the paper
- the area on the paper that the printer can use to format the document, which is usually smaller than the physical sheet of paper to account for margins or the mechanical limitations of the printer

The **page rectangle** represents the boundaries of the printable area on a page. Its upper-left corner always has the coordinates (0,0). The coordinates of the lower-right corner give the maximum page height and width attainable on the given printer; these coordinates are specified by the units used to express the resolution of the printing graphics port. For example, the lower-right corner of a page rectangle used by the PostScript LaserWriter printer driver for an 8.5-by-11-inch U.S. letter page is (730,552) at 72 dpi.

The **paper rectangle** gives the physical paper size, defined in the same coordinate system as the page rectangle. Thus, the upper-left coordinates of the paper rectangle are typically negative, and its lower-right coordinates are greater than those of the page rectangle. Figure 9-6 shows the relationship of these two rectangles.

Figure 9-6 Page and paper rectangles



Your application should always use the page rectangle sizes provided by the printer driver and should not attempt to change them or add new ones. If your application offers page sizes other than those provided by the printer driver for the current printer, you risk compatibility problems.

When formatting a page for printing, remember to use the page rectangle size that the user has chosen to format the document. (See "The TPrint Record and the Printing Loop" on page 9-11 for more information about where to find the user's choices for formatting the document.)

Printer Resolution

Resolution refers to the degree of detail at which a device such as a printer or a screen can display an image. Resolution is usually specified in dots per inch, or *dpi*, in the x and y directions. The higher the value, the finer the detail of the image.

A printer driver supports either discrete or variable resolution. If a printer driver supports **discrete resolution,** an application can choose from only a limited number of resolutions that are predefined by the printer driver. For example, the ImageWriter printer driver supports four discrete resolutions: 72 by 72 dpi, 144 by 144 dpi, 80 by 72 dpi, and 160 by 144 dpi.

If a printer driver supports **variable resolution,** an application can define any resolution within a range bounded by maximum and minimum values defined by the printer driver. LaserWriter printer drivers support variable resolution within a range from 25 dpi to 1500 dpi in both the x and y directions.

To print, your application does not need to check the resolutions available or set the resolution. However, if your application does factor in possible resolutions, it can obtain the best quality output from a printer by choosing equal resolutions for the x and y directions. For information on how to determine the available resolution or resolutions for the currently selected printer, see "Determining and Setting the Resolution of the Current Printer" on page 9-30.

The TPrint Record and the Printing Loop

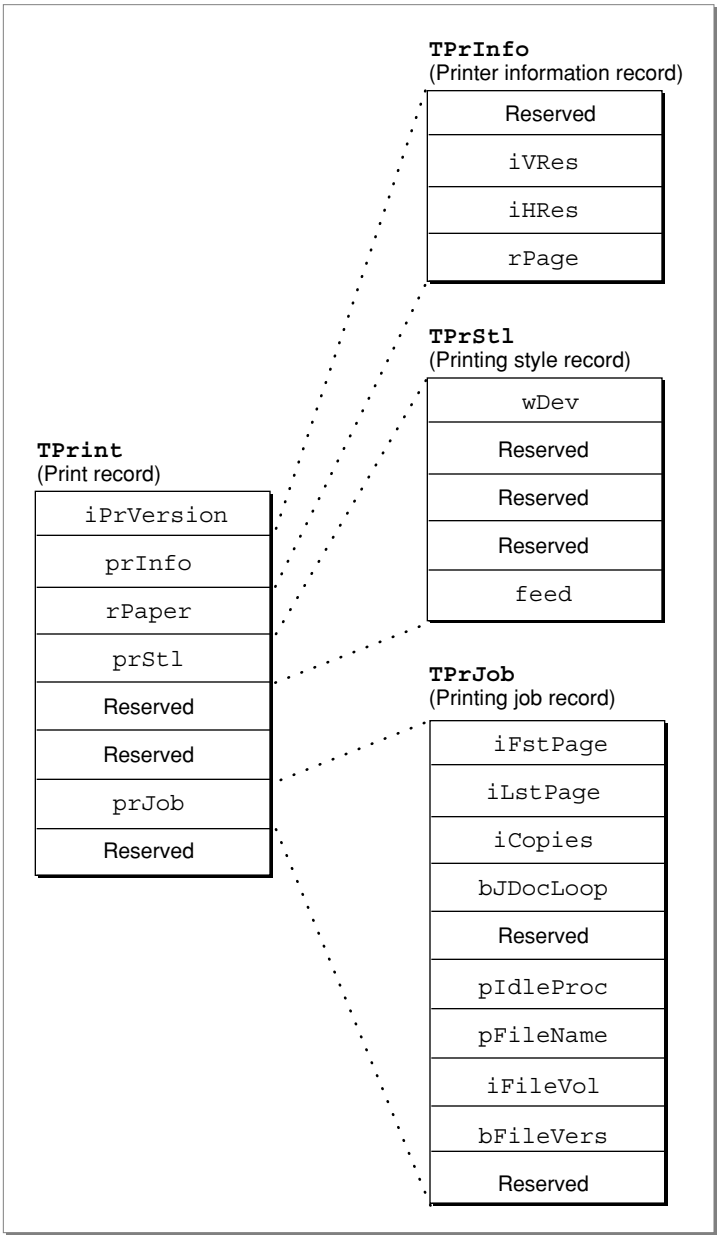
To print a document, you need to create a print record. The **`TPrint` record** is a data structure of type `TPrint`, and it contains fields that specify the Printing Manager version, information about the printer (such as its resolution in dpi), and the dimensions of the paper rectangle. Most Printing Manager routines require that you provide a handle to a `TPrint` record as a parameter.

Your application allocates the memory for a `TPrint` record (using the Memory Manager function `NewHandle`, for example) and then initializes the new `TPrint` record using the `PrintDefault` procedure. (Your application can also validate an existing `TPrint` record by using the `PrValidate` function.) When the user chooses the Print command, your application passes a handle to a `TPrint` record to the `PrJobDialog` or `PrDlgMain` function to display a job dialog box to the user; the function alters the `TPrint` record according to the user's responses.

When the user chooses the Page Setup command, your application passes a handle to a `TPrint` record to the `PrStlDialog` or `PrDlgMain` function to display a style dialog box to the user; the function alters the `TPrint` record according to the user's responses.

The `TPrint` record contains several other records, as illustrated in Figure 9-7. The `TPrInfo` record, which is a data structure of type `TPrInfo`, gives you the information needed for page composition, including the vertical and horizontal resolutions of the printer in dpi and the boundaries of the page rectangle. The **`TPrJob` record,** which is a data structure of type `TPrJob`, gives you information about a particular print job—for instance, the first and last pages to be printed, the number of copies, and the method of printing that the Printing Manager will use.

Figure 9-7 A `TPrint` record



The `PrJobDialog`, `PrStlDialog`, and `PrDlgMain` functions alter the appropriate fields of the `TPrint` record. In particular, the `PrJobDialog` function alters the `prJob` field (which contains a `TPrJob` record), and the `PrStlDialog` function alters the `prInfo` field (which contains a `TPrInfo` record). The `PrDlgMain` function alters either field, depending on whether you use the function to display a job or a style dialog box.

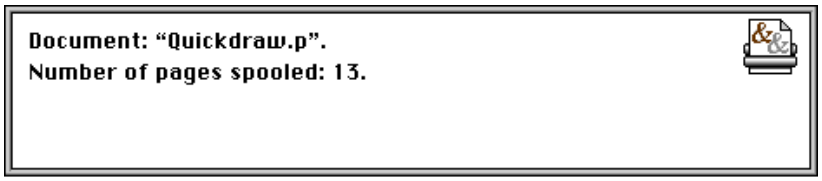
▲ **WARNING**
Your application should not directly change the user-supplied data in the `TPrint` record; it should use the `PrStlDialog` and `PrJobDialog` functions or the `PrDlgMain` function to allow the user to specify printing options that the printer driver then translates to the appropriate fields in the `TPrint` record. The only fields you may need to set directly are those containing optional information in the `TPrJob` record (for example, the `pIdleProc` field, which contains a pointer to an idle procedure). Attempting to set other values directly in the `TPrint` record can produce unexpected results. ▲

Your program code should contain a **printing loop** that handles your printing needs, including presenting the job dialog box and determining the range of pages to be printed. An example of a printing loop is shown in Listing 9-2 on page 9-20; the structure of a `TPrint` record is defined in detail on page 9-44.

Print Status Dialog Boxes

Because the user must wait for a document to print (that is, the application must draw the data in the printing graphics port and the data must be sent either to the printer or to a spool file before the user can continue working), many printer drivers display a print status dialog box informing the user that the printing process is under way. As shown in Figure 9-8, the print status dialog box usually provides information about the document being printed and indicates the current status of the printing operation.

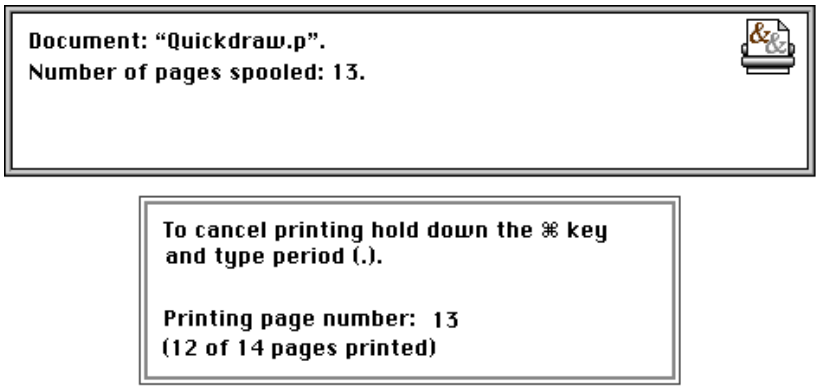
Figure 9-8 The print status dialog box for a LaserWriter printer driver printing in the background



A user should always be able to cancel printing by pressing Command-period. To determine whether the user has canceled printing, the printer driver runs an **idle procedure** whenever it directs output to the printer or to a spool file. The idle procedure takes no parameters and returns no result; the printer driver runs it periodically.

Not all printer drivers, however, display print status dialog boxes. As you can see in Figure 9-8, the print status dialog box for the LaserWriter printer driver (as well as status dialog boxes for many other printer drivers) doesn't inform the user how to cancel the printing operation. Your application can display its own status dialog box that informs the user about the status of the printing operation and how to cancel printing. If the printer driver displays its own print status dialog box, your application's print status dialog box may appear in an inactive window. Figure 9-9 shows an example of an application's print status dialog box that appears in addition to the print status dialog box displayed by the LaserWriter printer driver.

Figure 9-9 A status dialog box with the LaserWriter printer driver's print status dialog box



Note
Your application cannot prevent a printer driver from displaying its own status dialog box, and your application cannot determine where on the screen a printer driver will display its status dialog box. ◆

If your application uses its own print status dialog box, your application should display it just before printing. Your print status dialog box should indicate that the user can press Command-period to cancel printing; your status dialog box may also provide a button that lets the user cancel printing. Your status dialog box should also give information about the document being printed and indicate the current status of the printing operation.

The `TPrJob` record contained in the `TPrint` record contains a pointer to an idle procedure in the `pIdleProc` field. If this field contains the value `NIL`, then the printer driver uses its default idle procedure. The default idle procedure checks for Command-period keyboard events and sets the `iPrAbort` error code if one occurs, so that your application can cancel the print job at the user's request. However, the default idle procedure does not display any dialog boxes. It is up to the printer driver or your application to display a print status dialog box.

To handle update information in your status dialog box during the printing operation, you should install your own idle procedure in the `pIdleProc` field. Your idle procedure should also check whether the user has pressed Command-period, in which case your application should stop its printing operation. If your status dialog box contains a button to cancel the printing operation, your idle procedure should also check for clicks in the button and respond accordingly.

IMPORTANT
In your status dialog box, do not include an option to pause the printing process. Pausing may cause timeout problems when printing to a printer on a network. Communication between the Macintosh computer and the printer must be maintained to prevent a job or a wait timeout. If there is no communication for a period of time (over two minutes, for example, for the PostScript LaserWriter printer), the printer times out and the print job terminates. Because there is no reliable method for determining the type of printer, you should be aware of the possibility of a printer on a network timing out for a user who wants to pause printing for over two minutes. ▲

If you do not supply your own idle procedure, you can determine whether the user has canceled printing by calling the `PrError` function after each call to a Printing Manager routine. The `PrError` function returns the result code `iPrAbort` when the user cancels printing.

See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information about creating and displaying dialog boxes. For information about creating an idle procedure, see "Writing an Idle Procedure" on page 9-38.

Using the Printing Manager

The Printing Manager defines routines that give your application device-independent control over the printing process. You can use these routines to print documents, to display and alter the print dialog boxes, and to handle printing errors.

To use the Printing Manager, you must first initialize QuickDraw, the Font Manager, the Window Manager, the Menu Manager, TextEdit, and the Dialog Manager. The first Printing Manager routine to call, when you are ready to print, is `PrOpen`; the last routine to call is `PrClose`.

All of the Printing Manager routines described in this chapter are available on both basic QuickDraw and Color QuickDraw systems using system software version 4.1 or later. However, not all printer drivers support all features provided by the `PrGeneral` procedure. (When you call the `PrGeneral` procedure, described in "Getting and Setting Printer Information" beginning on page 9-28, it in turn calls the current printer driver to get or set the desired information.) After calling `PrGeneral` and passing it a particular opcode, you should call the `PrError` function and test whether it returns the `opNotImpl` result code, which indicates that the printer driver does not support that particular opcode.

All printable documents must have a `TPrint` record. Each `TPrint` record contains information about page size, number of copies requested, and the range of pages the user wants printed. Although only the information the user specifies through the style dialog box should be preserved each time the user prints the document, you can save the entire `TPrint` record when you save the document. The next time the user opens the document, you can retrieve the user's preferences as saved in the `TPrint` record and then use the `PrValidate` function to validate the fields of the `TPrint` record.

To print a user's document, you must first create or validate a `TPrint` record for the document. You can use the `PrintDefault` procedure to initialize the values in a `TPrint` record. You can use the `PrValidate` function to check that an existing `TPrint` record is compatible with the current printer and its driver. Your application should include a printing loop that handles printing and checks for printing errors at every step.

You should never assume the type of printer that has been selected; your application should always be able to print to any type of printer. However, for some special features that are not supported by QuickDraw (notably rotated text and graphics, dashed lines, and hairlines), you may want to create two versions of your drawing code: one that uses picture comments to take advantage of the features, and another that provides QuickDraw-based implementations of these features. Using picture comments, your application can instruct printer drivers to perform operations that QuickDraw does not support. For more information, see Appendix B, "Using Picture Comments for Printing," in this book.

The rest of this section describes how you can

- create and use a `TPrint` record
- structure your printing loop to print a document
- use the `PrGeneral` procedure to determine printer characteristics
- alter the style and job dialog boxes
- write an idle procedure that runs during printing
- handle printing errors

Be aware that the burden of maintaining backward compatibility with early Apple printer models—as well as maintaining compatibility with over a hundred existing printer drivers—requires extra care on your part. When the Printing Manager was initially designed, it was intended to support ImageWriter printers directly attached to Macintosh computers with only a single floppy disk and 128 KB of RAM. Later, the Printing Manager was implemented on PostScript LaserWriter printer drivers for more powerful Macintosh computers sharing LaserWriter printers on networks. Since then, the Printing Manager has been implemented on a substantial—and unanticipated—number of additional Apple and third-party printer drivers, each in its own, slightly unique way. When you use Printing Manager routines and data structures, you should be especially wary of and defensive about possible error conditions. Because Apple has little control over the manner in which third parties support the Printing Manager in their printer drivers, you should test your application's printing code on as many printers as possible.

Creating and Using a TPrint Record

To print a document, you need a valid `TPrint` record that is formatted for the current versions of the Printing Manager and the printer driver.

To create a new `TPrint` record, you must first create a handle to it with a Memory Manager function such as `NewHandle` or `NewHandleClear`. You then must use the `PrintDefault` procedure to set the fields of the record to the default values for the current printer driver, as illustrated in the following code fragment.

```
VAR
   prRecHdl:   THPrint;

   {allocate handle to a TPrint record}
   prRecHdl := THPrint(NewHandleClear(SizeOf(TPrint)));
   IF prRecHdl <> NIL THEN
      PrintDefault(prRecHdl) {sets appropriate default values }
                             { for current printer driver}
   ELSE
   ; {handle error here}
```

You can also use an existing `TPrint` record (for instance, one saved with a document). If you use an existing `TPrint` record, be sure to call the `PrValidate` function before using the `TPrint` record to make sure it's valid for the current version of the Printing Manager and for the current printer driver.

Listing 9-1 shows an application-defined routine that reads a `TPrint` record that the application has saved as a resource of type `'SPRC'` with the document. (The Resource Manager routines `CurResFile`, `UseResFile`, `Get1Resource`, and `DetachResource` that are shown in this listing are described in the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*.)

Listing 9-1 Reading a document's `TPrint` record

```
FUNCTION MyGetPrintRecordForThisDoc (refNum: Integer;
                                     VAR prRecHdl: THPrint;
                                     VAR prRecChanged: Boolean):
                                     OSErr;
VAR
   saveResFile:   Integer;
BEGIN
   saveResFile := CurResFile; {save the resource file for the document}
   UseResFile(refNum);
   prRecHdl := THPrint(Get1Resource('SPRC', kDocPrintRec));
   IF prRecHdl <> NIL THEN
```

```
   BEGIN
      DetachResource(Handle(prRecHdl));
      prRecChanged := PrValidate(prRecHdl);  {validate TPrint record}
      MyGetPrintRecordForThisDoc := PrError;
   END
   ELSE
      MyGetPrintRecordForThisDoc := kNILHandlePrintErr;
   UseResFile(saveResFile);
END;
```

You should save the `TPrint` record when the user saves the document. By doing this, you can save any preferences that the user has selected for printing that document, such as orientation of the page or page size. See the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for information about saving data such as `TPrint` records in resources.

Every printer driver uses the fields of the `TPrint` record differently. To maintain compatibility with the Printing Manager, you should follow these guidelines:

- Do not test for the contents of undocumented fields.
- Do not set fields in the `TPrint` record directly.
- Use the print dialog boxes provided by the printer drivers or, if you want to customize these dialog boxes, alter them only as recommended in "Altering the Style or Job Dialog Box" on page 9-35.

Printing a Document

When writing an application, the code you provide that handles printing is referred to as the *printing loop*. A printing loop calls all the Printing Manager routines necessary to print a document. In general, a printing loop must do the following tasks:

- It must unload unused code segments to ensure that you have as much memory as possible in which to print.
- It must open the Printing Manager and the current printer driver by using the `PrOpen` procedure.
- It must set up a valid `TPrint` record for the document (using any values the user previously specified through the style dialog box) by using the `PrintDefault` procedure or the `PrValidate` function. (When the user is printing from the Finder, it is best not to display the style dialog box, but rather to use saved or default settings for the document.)

- It must display the job dialog box as appropriate by using the `PrJobDialog` function or, for a customized job dialog box, the `PrDlgMain` function. (When the user is printing from the Finder, display the job dialog box only once, and then use the `PrJobMerge` procedure to apply the information from this dialog box to any other documents selected by the user.)
- It must determine the number of pages required to print the requested range of pages by examining the fields of the `TPrint` record. (Depending on the page rectangle of the current printer, the amount of data you can fit on a physical page of paper may differ from that displayed on the screen, although it is usually the same.)
- It must determine the number of copies to print by examining the `TPrint` record.
- It may display a status dialog box indicating to the user the status of the current printing operation by using the Dialog Manager function `GetNewDialog` (described in the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*).
- If it displays a status dialog box, it must install an idle procedure in the `pIdleProc` field of the `TPrJob` record (which is contained in the `TPrint` record) to update information in the status dialog box and to check whether the user wants to cancel the operation. (The default idle procedure also performs this check, but if you update information in your status dialog box you must provide your own idle procedure.)
- It must print the requested range of pages for each requested copy by
  - using the `PrOpenDoc` function to open a printing graphics port if the current page number is the first page or a multiple of the value represented by the constant `iPFMaxPgs`
  - opening a page for printing by using the `PrOpenPage` procedure
  - printing the page by drawing into the printing graphics port with the QuickDraw routines described in the rest of this book
  - closing the page by using the `PrClosePage` procedure
  - using the `PrCloseDoc` procedure to close the printing graphics port and begin printing the requested range of pages
  - checking whether the printer driver is using deferred printing and, if so, using the `PrPicFile` procedure to send the spool file to the printer
- Finally, the printing loop must close the Printing Manager by using the `PrClose` procedure.

Listing 9-2 shows an extremely broad example of a printing loop—the code does not optimize for the type of printer being used or for the material being printed (text, graphics, or a mixture of both). However, this sample routine, called `MyPrintLoop`, does cover the major aspects of a printing loop: how to balance calls to the open and close routines, how to determine page count, and how to provide support for documents exceeding the maximum page length specified by the constant `iPFMaxPgs`.

Listing 9-2 A sample printing loop

```
PROCEDURE MyPrintLoop(docToPrint: MyDocRecHnd; displayJob: Boolean);
VAR
   copies, numberOfCopies:    Integer;
   firstPage, lastPage:       Integer;
   pageNumber, numberOfPages: Integer;
   doPrint, changed:          Boolean;
   oldPort:                   GrafPtr;
   theStatus:                 TPrStatus;
   printError:                Integer;
BEGIN
   GetPort(oldPort);
   MyUnLoadTheWorld; {swap out those segments of code not needed to print}
   PrOpen;           {open Printing Manager and the current printer driver}
   IF (PrError = noErr) THEN
   BEGIN
      gPrintResFile := CurResFile;     {save the current resource file}
      gPrintRec := docToPrint^^.docPrintRecHdl; {set to this doc's print rec}
      changed := PrValidate(gPrintRec);   {verify TPrint record}
      IF (PrError = noErr) THEN
      BEGIN
         {determine the number of pages required to print the document}
         numberOfPages := MyDetermineNumOfPages(gPrintRec^^.prInfo.rPage);
         {display job dialog box if requested, else use previous settings}
         IF displayJob THEN
            doPrint := PrJobDialog(gPrintRec)
         ELSE
            doPrint := MyDoJobMerge(gPrintRec);
         IF doPrint THEN
         BEGIN
            numberOfCopies := gPrintRec^^.prJob.iCopies;
            firstPage := gPrintRec^^.prJob.iFstPage;{save first page number}
            lastPage := gPrintRec^^.prJob.iLstPage; {save last page number}
            gPrintRec^^.prJob.iFstPage := 1;        {reset to 1}
            gPrintRec^^.prJob.iLstPage := iPrPgMax; {reset to maximum}
```

```
IF (numberOfPages < lastPage) THEN
   lastPage := numberOfPages;    {to prevent printing past last }
                                 { page}
{display a "Print Status" dialog box (optional)-- }
{ first, deactivate front window}
MyDoActivateFrontWindow(FALSE, oldPort);
gPrintStatusDlg := GetNewDialog(kPrintStatus, NIL, Pointer(-1));
{set up dialog items (insert name of document being printed)}
MySetUpDBoxItems(docToPrint);
ShowWindow(gPrintStatusDlg);  {display the dialog box}
{set up idle procedure (for later use)}
gPrintRec^^.prJob.pIdleProc := @MyDoPrintIdle;
{print the requested number of copies}
FOR copies := 1 TO numberOfCopies DO
BEGIN
   UseResFile(gPrintResFile);{restore driver's resource file}
   {print the requested range of pages of the document}
   FOR pageNumber := firstPage TO lastPage DO
   BEGIN
      {check current page number against iPFMaxPgs}
      IF (pageNumber - firstPage) MOD iPFMaxPgs = 0 THEN
      BEGIN
         IF pageNumber <> firstPage THEN
         {if max size of spool file has been reached (and this }
         { isn't the first page), then close the document, }
         { initiate printing, then reopen the document}
         BEGIN
            PrCloseDoc(gPrinterPort);
            {next line tests for deferred printing}
            IF (gPrintRec^^.prJob.bJDocLoop = bSpoolLoop)
                  AND (PrError = noErr) THEN
               PrPicFile(gPrintRec, NIL, NIL, NIL, theStatus);
         END;
         {if this is the first page or a multiple of iPFMaxPgs, }
         { then open the document for printing}
         gPrinterPort := PrOpenDoc(gPrintRec, NIL, NIL);
      END; {of check current page number}
      IF (PrError = noErr) THEN
      BEGIN {print a page}
         PrOpenPage(gPrinterPort, NIL);
         IF (PrError = noErr) THEN
         {draw (print) a page in the printable area for the }
         { current printer (indicated by the rPage field)}
```

```
                  MyDrawStuff (gPrintRec^^.prInfo.rPage, docToPrint,
                               GrafPtr(gPrinterPort), pageNumber);
                  PrClosePage(gPrinterPort);
               END;  {of print a page}
            END; {of print the requested range of pages}
            PrCloseDoc(gPrinterPort);
            IF (gPrintRec^^.prJob.bJDocLoop = bSpoolLoop) AND
               (PrError = noErr) THEN
               PrPicFile(gPrintRec, NIL, NIL, NIL, theStatus);
         END;
      END;
   END;
END;
```

The `MyPrintLoop` procedure starts by getting a pointer to the current graphics port. Then it calls an application-defined routine, `MyUnloadTheWorld`, that swaps out code segments not required during printing. Then it opens the Printing Manager and the current printer driver and its resource file by calling `PrOpen`.

The `MyPrintLoop` procedure saves the current resource file (after calling `PrOpen`, the current resource file is the driver's resource file) so that, if its idle procedure changes the resource chain in any way, it can restore the current resource file before returning; thus the driver does not lose access to its resources. The `MyPrintLoop` procedure then uses the `PrValidate` function to change any values in the `TPrint` record associated with the document to match those specified by the current printer driver; these values can be changed later by the printer driver as a result of your application's use of the `PrStlDialog` and `PrJobDialog` functions. (Your application passes a handle to a `TPrint` record to the `PrStlDialog` and `PrJobDialog` functions, and these procedures modify the `TPrint` record according to the user's interaction with the style and job dialog boxes.) The `MyPrintLoop` procedure calls `PrValidate` rather than `PrintDefault` to preserve any values that the user might have previously set through the style dialog box.

To print a document, you must divide the data into sections that fit within the page rectangle dimensions stored in the `rPage` field of the `TPrJob` record, which is contained in the `TPrint` record. (This information is stored in the `rPage` field when you call the `PrintDefault`, `PrValidate`, or `PrStlDialog` routine.) The application-defined function `MyDetermineNumOfPages` is specific to the application, because the way the application divides up the data depends on the type of text and graphics in the document. The `MyDetermineNumOfPages` function determines the number of pages required to print the document by comparing the size of the document with the printable area for the current printer, which is specified by the value in the `rPage` field of the `TPrJob` record in the `TPrint` record.

After determining the number of pages required to print the document, `MyPrintLoop` displays the job dialog box if the calling routine requested it to do so. If the user prints multiple documents at once, the calling routine sets the `displayJob` parameter to `TRUE` for the first document and `FALSE` for subsequent documents. This allows the user to specify values in the job dialog box only once when printing multiple documents. It also provides an application with the ability to print documents in the background (for example, as the result of responding to the Apple event Print Documents) without requiring the application to display the job dialog box.

The user's responses in the job dialog box provide such information as the number of copies and the page numbers of the first and last pages requested. The `MyPrintLoop` procedure stores these values in the local variables `firstPage` and `lastPage`. It then resets the value of the first page in the `TPrJob` record as 1 and resets the value of the last page to the value represented by the constant `iPrPgMax`.

The `MyPrintLoop` procedure compares the values of the number of pages in the document with the last page the user requested and changes the last page number as necessary. For example, if the user asks to print page 50 of a two-page document, `MyPrintLoop` resets the value of the last page to 2.

At this point, `MyPrintLoop` is about to begin the process of sending the pages off to be printed. So it displays its own status dialog box to inform the user of the current status of the printing operation. If your status dialog box provides a button or reports on the progress of the printing operation, you need to handle events in the dialog box by providing an idle procedure. Your idle procedure should update the items in your status dialog box to show the current progress of the printing operation, and it should determine whether the user has canceled the printing operation. The printer driver calls the idle procedure periodically during the printing process. For more information on idle procedures, see "Writing an Idle Procedure" on page 9-38.

After installing its idle procedure, the `MyPrintLoop` procedure then begins the printing operation by performing a number of steps for each requested copy. First, `MyPrintLoop` restores the current resource file to the printer driver's resource file.

`MyPrintLoop` then begins the process of printing each page. The maximum number of pages that can be printed at a time is represented by the constant `iPFMaxPgs`. If the file is larger than the value represented by `iPFMaxPgs`, your application can print the number of pages represented by `iPFMaxPgs` and then begin the printing loop again with the next section of the document. In this way, you can print any number of pages.

Next, `MyPrintLoop` opens a page for printing and draws the page in the printing graphics port with the application-defined `MyDrawStuff` procedure, the details of which are specific to the application. The parameters to `MyDrawStuff` are the size of the page rectangle, the document containing the data to print, the printing graphics port in which to draw, and the page number to be printed. This allows the application to use the same code to print a page of a document as it uses to draw the same page on screen.

When `MyPrintLoop` is finished printing (or has printed a multiple of the value represented by the constant `iPFMaxPgs`), it closes the printing graphics port for the document. By testing for the `bSpoolLoop` constant in the `bJDocLoop` field of the `TPrJob` record, `MyPrintLoop` determines whether a printer driver is using deferred printing; if so, `MyPrintLoop` calls the `PrPicFile` procedure, which sends the spool file to the printer.

Some QuickDraw printer drivers (in particular, those for the ImageWriter and ImageWriter LQ printers) provide two methods of printing documents: deferred and draft-quality. Typically, the printer driver uses deferred printing when a user chooses Best in the job dialog box, and it uses draft-quality printing when the user chooses Draft.

Deferred printing was designed to allow ImageWriter printers to spool a page image to disk when printing under the low memory conditions of the original 128 KB Macintosh computer. With **deferred printing,** a printer driver records each page of the document's printed image in a structure similar to a QuickDraw picture, which the driver writes to a spool file. For compatibility with printer drivers that still support deferred printing, use the `PrPicFile` procedure to instruct these printer drivers to turn the QuickDraw pictures into bit images and send them to the printer. (**Draft-quality printing,** on the other hand, is a method by which a printer driver converts into drawing operations calls only to QuickDraw's text-drawing routines. The printer driver sends these routines directly to the printer instead of using deferred printing to capture the entire image for a page in a spool file.)

Note
Do not confuse background printing with deferred printing. While printer drivers supporting background printing also create spool files, you do not need to use the `PrPicFile` procedure to send these spool files to the printer. In fact, there is no reliable way for you to determine whether a printer driver is using a spool file for background printing. ◆

The `MyPrintLoop` procedure concludes by closing the Printing Manager, reporting any Printing Manager errors, and resetting the current graphics port to the original port.

In your printing loop, you should balance all calls to Printing Manager open routines to the equivalent Printing Manager close routines. This is extremely important, even if you stop printing because of an error. Failure to call the matching close routines can cause the Printing Manager to perform incorrectly.

Note that `MyPrintLoop` calls `PrError` after each Printing Manager routine. If an error is found, the loop calls a close routine (`PrClose`, `PrClosePage`, or `PrCloseDoc`) for any Printing Manager open routines (`PrOpen`, `PrClosePage`, or `PrOpenDoc`) before informing the user of the error. You should use this approach in your own application to make sure the Printing Manager closes properly and all temporary memory is released.

▲ **WARNING**
Some applications use a method of printing that prints out each page of a spooled document as a separate print job in order to avoid running out of disk space while spooling the document. You should not use this method, known as "spool a page, print a page." It is appropriate only for a printer directly connected to the user's computer (that is, not to a network) and therefore creates device dependence—and also it's extremely slow. If the printer is a remote or shared device (such as a LaserWriter printer connected by an AppleTalk network), another application could print a document between the pages of your user's document. At worst, if both applications printing to the shared printer use the "spool a page, print a page" method, the printed documents could end up interleaved. The pages for one of the documents could be out of order, even when printed by itself on a shared, network printer. ▲

Printing From the Finder

Typically, users print documents that are open on the screen one at a time while the application that created the document is running. Alternatively, users can print one or more documents from the Finder. To print documents from the Finder, the user selects one or more document icons and chooses the Print command from the File menu. When the Print command is chosen, the Finder starts up the application and passes it an Apple event—the Print Documents event—indicating that the documents are to be printed rather than opened on the screen.

As explained in *Inside Macintosh: Interapplication Communication*, your application should support the required Apple events, which include the Print Documents event. In response to a Print Documents event, your application should do the following:

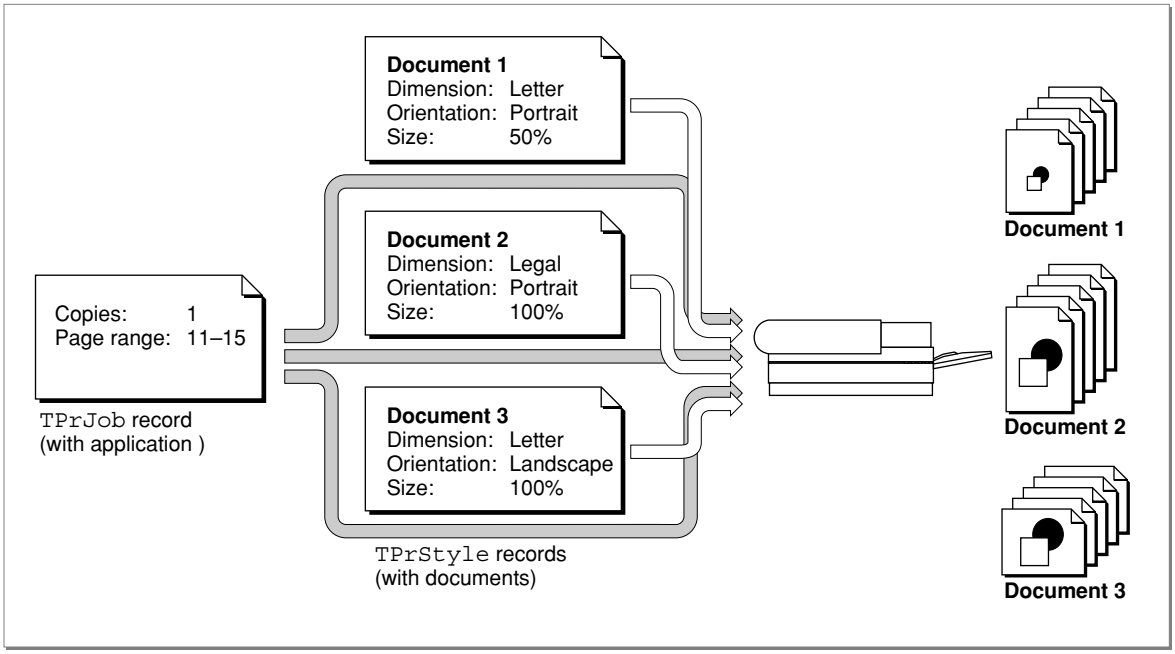
1. Your application should not open windows for the documents.
2. For style information, your application should use saved or default settings instead of displaying the style dialog box to ask this information from the user.

3. Your application should use the `PrJobDialog` function (described on page 9-62) or the `PrDlgMain` function (described on page 9-63) to display the job dialog box only once. When the user clicks the OK button in the job dialog box, you can then use the `PrJobMerge` procedure (described on page 9-66) to apply the information specified by the user to all of the documents selected from the Finder.

   For example, if the user has selected three documents to print, you can display the job dialog box only once and then apply the same information supplied by the user to all three documents. Figure 9-10 shows a situation where, through the job dialog box, the user has specified the number of copies and the range of pages to print. In this example, the application applies this job information to the `TPrint` record of the three documents by calling `PrJobMerge`. Note that `PrJobMerge` preserves the fields of the `TPrint` record that are specific to each document (that is, the fields that are set by the user through the style dialog box).

4. Your application should remain open until the Finder sends your application a Quit event. If appropriate, the Finder sends your application this Apple event immediately after sending it the Print Documents event.

Figure 9-10 How the `PrJobMerge` procedure works



See *Inside Macintosh: Interapplication Communication* for more information about how to handle the Print Documents and Quit events.

Providing Names of Documents Being Printed

Some printer drivers (usually those for printers such as LaserWriter printers that are shared among many users) provide the names of the users who are printing and the documents that are being printed to others interested in using the printer. Providing the names of users and documents is a courtesy to other users sharing the printer on a network. The printer driver gets the name of the document being printed from the title of the frontmost window on the user's screen. The `PrOpenDoc` and `PrValidate` functions call the Window Manager procedure `FrontWindow` to get the document's name.

Printer drivers can't get a document name if your application doesn't display windows while printing. For example, applications should not open windows for their documents when the user prints from the Finder. If there is no front window, or if the window's title is empty, the printer driver sets the document name to "Unspecified" or "Untitled."

You can ensure that the document name is available by displaying a printing status dialog box and setting the window's title to the document's name. If the dialog box is one that doesn't have a title bar (like that of type `dBoxProc`), this title is not displayed but the current printer driver can still use the title as the document's name. If you don't want to put up a visible window, you can create a tiny window (for instance, type `plainDBox`) and hide it behind the menu bar by giving it the global coordinates of (1,1,2,2). See the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information about the `dBoxProc` and `plainDBox` window types.

Note
Do not set the document name in the `TPrint` record directly. Not all printer drivers support this field, and Apple does not guarantee that internal fields of the Printing Manager's data structures will remain the same. ◆

Printing Hints

QuickDraw is the primary means you use to print, and in general you can use QuickDraw in the printing graphics port exactly as you would for a screen's graphics port. There are a few things to note when drawing to the printing graphics port:

- Don't depend on values in a printing graphics port remaining identical from page to page. With each new page, you usually get reinitialized font information and other characteristics for the printing graphics port.
- Don't make calls that don't do anything on the printer. For example, QuickDraw erase routines such as `EraseRect` are quite time-consuming and normally aren't needed on the printer. An erase routine takes time because every bit (90,000 bits per square inch on a 300 dpi LaserWriter) has to be cleared. Paper does not need to be erased the way the screen does. Also avoid using the `TextBox` procedure, which makes calls to the `EraseRect` procedure. You might want to use a different method of displaying text (for example, `DrawString` or `DrawText`) or write your own version of `TextBox`. See the chapter "QuickDraw Text" in *Inside Macintosh: Text*.

- Don't use clipping to select text to be printed. There are a number of subtle differences between how text appears on the screen and how it appears on the printer; you can't count on knowing the exact dimensions of the rectangle occupied by the text.
- Don't use fixed-width fonts to align columns. Because spacing is adjusted on the printer, you should explicitly move the pen to where you want it.
- Don't use the outline font style to create white text on a black background.
- Avoid changing fonts frequently.
- Because of the way rectangle intersections are determined, you slow printing substantially if your clipping region falls outside of the rectangle given by the `rPage` field of the `TPrInfo` record of the `TPrint` record.

Getting and Setting Printer Information

You can determine the resolution of the printer, set the resolution you want, find out if the user has selected landscape printing, or force enhanced draft-quality printing by using the `PrGeneral` procedure. You call the `PrGeneral` procedure with one of five opcodes: `getRslDataOp`, `setRslOp`, `getRotnOp`, `draftBitsOp`, or `noDraftBitsOp`. These opcodes have data structures associated with them.

When you call the `PrGeneral` procedure, it in turn calls the current printer driver to get or set the desired information. Not all printer drivers support all features provided by the `PrGeneral` procedure, however, so your application can't depend on its use.

Listing 9-3 shows an application-defined routine, `DoIsPrGeneralThere`, that checks whether the current printer driver supports the `PrGeneral` procedure. First, `DoIsPrGeneralThere` sets the `opcode` field of the `TGetRotnBlk` record to the `getRotnOp` opcode—the opcode used to determine whether the user has chosen landscape orientation. Then `DoIsPrGeneralThere` passes the address of the `TGetRotnBlk` record to the `PrGeneral` procedure. It then calls `PrError` to get any errors that result from calling `PrGeneral`. If the error is `resNotFound`, the printer driver does not support `PrGeneral`.

Listing 9-3 Checking whether the current printer driver supports the `PrGeneral` procedure

```
FUNCTION DoIsPrGeneralThere: Boolean;
VAR
   getRotRec:     TGetRotnBlk;
   myPrintErr:    OSErr;
BEGIN
   myPrintErr := 0;
   getRotRec.iOpCode := getRotnOp;  {set the opcode}
   getRotRec.hPrint := gMyPrRecHdl; {TPrint record this operation applies to}
   PrGeneral(@getRotRec);
   myPrintErr := PrError;
   PrSetError(noErr);
   IF (myPrintErr = resNotFound) THEN  {the current driver doesn't support }
      DoIsPrGeneralThere := FALSE;     { PrGeneral}
   ELSE
      DoIsPrGeneralThere := TRUE;      {current driver supports PrGeneral}
END;
```

After determining that the current printer driver supports `PrGeneral`, you can use `PrGeneral` to

- determine and set the resolution of the current printer
- determine the current page orientation
- force enhanced draft-quality printing

As an alternative to testing for `PrGeneral`, your application can call `PrGeneral` and then test whether `PrError` error returns the `opNotImpl` result code, which indicates that the printer driver either does not support `PrGeneral` or does not support that particular opcode.

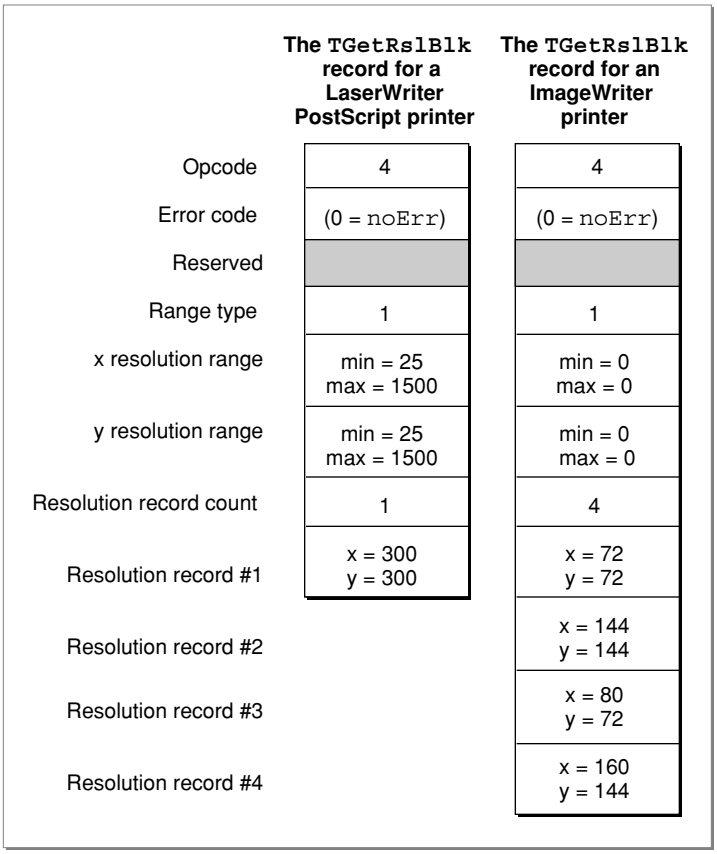
These operations are discussed in the following sections.

Determining and Setting the Resolution of the Current Printer

Some printer drivers support only one of the two possible kinds of resolution: discrete or variable. You can use the `PrGeneral` procedure to determine the kind of resolution supported by the current printer and then use the highest resolution desired by your application or the user.

Each printer has its own imaging capabilities. When you call `PrGeneral` with the value `getRslDataOp` in the `iOpCode` field of the `TGetRslBlk` record, `PrGeneral` returns the resolutions that the printer supports. Figure 9-11 shows `TGetRslBlk` records (described on page 9-53) returned by the drivers for a 300-dpi LaserWriter PostScript printer and a QuickDraw ImageWriter printer. Because it supports variable resolutions, the `TGetRslBlk` record for the LaserWriter driver specifies minimum and maximum resolutions in the x and y directions. Because it uses discrete resolutions, the `TGetRslBlk` record for the ImageWriter driver specifies no minimum or maximum resolutions in the x and y directions, but instead specifies the four discrete resolutions it supports.

Figure 9-11 Sample resolutions for a PostScript printer and a QuickDraw printer



A TPrint record contains the x and y resolutions that the printer uses in printing the data associated with the TPrint record. For each TPrint record you use, you can either use the default values or you can specify the particular imaging resolution that you want to use. To do this, you can call PrGeneral, specifying the value setRslOp in the iOpCode field and specifying the x and y resolutions in the iXRsl and iYRsl fields of the TSetRslBlk record (which is described on page 9-54). The PrGeneral procedure returns the noErr result code if it has updated the TPrint record with this new resolution, or it returns the noSuchRsl result code if the current printer doesn't support this resolution.

Listing 9-4 illustrates how to use the PrGeneral procedure to determine the possible resolutions for the current printer and then set a TPrint record to the desired resolution.

Listing 9-4 Using the getRslDataOp and setRslOp opcodes with the PrGeneral procedure

```
FUNCTION DoSetMaxResolution (thePrRecHdl: THPrint): Integer;
VAR
   maxDPI:     Integer;
   resIndex:   Integer;
   getResRec:  TGetRslBlk;
   setResRec:  TSetRslBlk;
BEGIN
   maxDPI := 0;
   getResRec.iOpCode := getRslDataOp;{get printer resolution info}
   PrGeneral(@getResRec);
   IF (getResRec.iError = noErr) AND (PrError = noErr) THEN
   BEGIN
      {the TGetRslBlk record contains an array of possible resolutions-- }
      { so loop through each resolution range record looking for }
      { the highest resolution available where x and y are equal}
      FOR resIndex := 1 TO (getResRec.iRslRecCnt) DO
      BEGIN
         IF (getResRec.rgRslRec[resIndex].iXRsl =
               getResRec.rgRslRec[resIndex].iYRsl) AND
               (getResRec.rgRslRec[resIndex].iXRsl > maxDPI) THEN
            maxDPI := getResRec.rgRslRec[resIndex].iYRsl;
      END;
      {set the resolution to the maximum supported resolution}
      IF maxDPI <> 0 THEN
      BEGIN
         WITH setResRec DO
         BEGIN
            iOpCode := setRslOp;
            hPrint := thePrRecHdl;
```

```
            iXRsl := maxDPI;
            iYRsl := maxDPI;
         END;
         PrGeneral(@setResRec);
      END; {end of maxDPI <> 0}
      IF (setResRec.iError = noErr) AND (PrError = noErr) AND
            (maxDPI <> 0) THEN
         DoSetMaxResolution := maxDPI;
   END
   ELSE
      DoSetMaxResolution := 0;
END;
```

You can reset the original resolutions by calling the `PrGeneral` procedure with the `setRslOp` opcode a second time. To do so, you should save the values contained in the `iVRes` and `iHRes` fields of the `TPrInfo` record before making the first call to `PrGeneral`. You can also reset the original resolutions by calling the `PrintDefault` procedure with the `TPrint` record, which sets all of the fields of the `TPrint` record to the default values of the current printer resource file. However, if you use `PrintDefault` you lose all of the user's selections from the last style dialog box. (You may want to reset the original resolution because that may be the printer's best resolution, though not its highest.)

Based on the information you get with a call to `PrGeneral` using the `getRslDataOp` opcode, you may decide to change the resolution with a call to `PrGeneral` using the `setRslOp` opcode. If so, the printer driver may need to change the appearance of the style and job dialog boxes by disabling some items. Therefore, you should determine and set the resolution *before* you use the `PrStlDialog` and `PrJobDialog` functions (or the `PrDlgMain` function) to present the print dialog boxes to the user.

Note that the style dialog boxes for some printers, such as the StyleWriter, may offer the user a choice of printing in Best or Normal modes, which sets the printing at 360 or 180 dpi, respectively. Your application has no control over this setting. The printer driver converts your drawing accordingly.

Determining Page Orientation

At times it can be useful for your application to determine which page orientation the user selects in the style dialog box. For instance, if an image fits on a page only if it is printed in landscape orientation (the `prInfo` field of the `TPrint` record defines a smaller horizontal value for the paper rectangle than for the image rectangle) and the user has not selected landscape orientation, your application can remind the user to select this orientation before printing. Otherwise, the user gets a clipped image.

If you call the `PrGeneral` procedure with the `getRotnOp` opcode in the `TGetRotnBlk` record (described on page 9-56), the printer driver returns in the `fLandscape` field of this record a Boolean variable that indicates whether or not the `TPrint` record specifies

landscape orientation. The user selects the type of orientation through the style dialog box, and the printer driver updates the fields of the `TPrint` record accordingly.

Listing 9-5 shows an application-defined function, `DoIsLandscapeModeSet`, that returns a Boolean value indicating whether the user has selected landscape orientation for the current document.

Listing 9-5 Using the `getRotnOp` opcode with the `PrGeneral` procedure to determine page orientation

```
FUNCTION DoIsLandscapeModeSet (thePrRecHdl: THPrint): Boolean;
VAR
   getRotRec:  TGetRotnBlk;
BEGIN
   getRotRec.iOpCode := getRotnOp;  {set opcode}
   getRotRec.hPrint := thePrRecHdl; {specify TPrint record}
   PrGeneral(@getRotRec);           {get landscape orientation}
   IF (getRotRec.iError = noErr) AND (PrError = noErr) AND
      getRotRec.fLandscape THEN
      DoIsLandscapeModeSet := TRUE
   ELSE
      DoIsLandscapeModeSet := FALSE;
END;
```

Enhancing Draft-Quality Printing

When the user selects faster, draft-quality printing from a job dialog box from some printer drivers, the printer driver handles the printing operation appropriately.

However, you can force users to use an enhanced form of draft-quality printing on ImageWriter printers (as well as on other printers that may support enhanced draft-quality printing) by calling the `PrGeneral` procedure, specifying the `draftBitsOp` opcode in a `TDftBitsBlk` record (described on page 9-55), and specifying the `TPrint` record for the operation. If your application produces only text, bitmaps, or pixel maps, this can increase performance and save disk space, because the printer driver prints the document immediately, rather than spooling it to disk as with deferred printing. The `draftBitsOp` opcode has no effect if the printer driver does not support draft-quality printing or does not support deferred printing. If the driver does not support the `draftBitsOp` opcode, the `PrGeneral` procedure returns the `opNotImpl` result code.

With draft-quality printing, a printer driver like the ImageWriter printer driver converts into drawing operations calls only to QuickDraw's text-drawing routines. The printer driver sends these text-drawing routines directly to the printer instead of using deferred printing to capture the entire image for a page in a spool file. Draft-quality printing produces quick, low-quality drafts of text documents that are printed straight down the page, from top to bottom and left to right.

Using the `PrGeneral` procedure, it's possible to produce enhanced draft-quality printing on some printers—such as ImageWriter printers. Normally, draft-quality printing renders output consisting only of text. However, **enhanced draft-quality printing** prints the bitmaps and pixel maps that your application draws using the `CopyBits` procedure (described in the chapter "QuickDraw Drawing" in this book) without using deferred printing to write to and read from a spool file.

Because it's supported by so few printer drivers, and because it offers little in the way of extra capability, enhanced draft-quality printing has limited usefulness.

To use enhanced draft-quality printing, call `PrGeneral` with the `draftBitsOp` opcode before using the `PrStlDialog` and `PrJobDialog` functions or the `PrDlgMain` function to present the style dialog box and job dialog box to the user. The use of the `draftBitsOp` opcode may cause items in the print dialog boxes to become inactive. For the ImageWriter printer driver, for example, the use of the `draftBitsOp` opcode makes the landscape icon in the style dialog box and the Best and Faster options in the job dialog box inactive.

IMPORTANT

If you call `PrGeneral` with the `draftBitsOp` opcode after using the `PrJobDialog` or `PrDlgMain` function, and if the user chooses draft printing from the job dialog box, the ImageWriter printer does not print any bitmaps or pixel maps contained in the document. ▲

Listing 9-6 illustrates how to implement enhanced draft-quality printing.

Listing 9-6 Using the `draftBitsOp` opcode with the `PrGeneral` procedure for enhanced draft-quality printing

```
FUNCTION DoDraftBits (thePrRecHdl: THPrint): Boolean;
VAR
   draftBitsBlk:  TDftBitsBlk;
BEGIN
   draftBitsBlk.iOpCode := draftBitsOp;{set the opcode}
   draftBitsBlk.hPrint := thePrRecHdl; {specify the TPrint record}
   PrGeneral(@draftBitsBlk);        {use enhanced draft quality}
   IF (draftBitsBlk.iError = noErr) AND (PrError = noErr) THEN
      DoDraftBits := TRUE     {this TPrint record specifies }
                              { enhanced draft printing}
   ELSE
      DoDraftBits := FALSE;   {this TPrint record does not }
                              { specify enhanced draft printing}
END;
```

You should keep one additional point in mind when using the `draftBitsOp` opcode: all of the data that is printed must be sorted along the y axis, because reverse paper motion is not possible on the ImageWriter printer when printing in draft-quality mode. This means that you cannot print two objects side by side; that is, the top boundary of an object cannot be higher than the bottom boundary of the previous object. To get around this restriction, you should sort your objects before print time.

You can call `PrGeneral` with the `noDraftBitsOp` opcode to use regular draft-quality printing again. If you call `PrGeneral` with `noDraftBitsOp` without first calling `draftBitsOp`, the procedure does nothing. As with the `draftBitsOp` opcode, you should call `PrGeneral` with the `noDraftBitsOp` opcode before you present the style and job dialog boxes to the user.

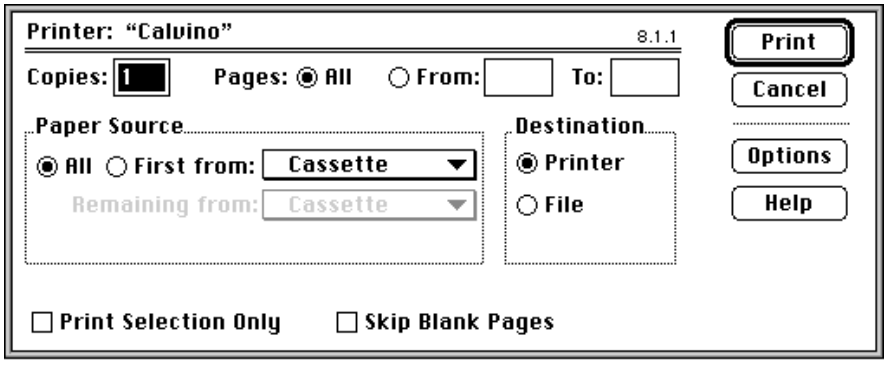
Altering the Style or Job Dialog Box

Each printer resource file includes definitions of the standard style and job dialog boxes that are specific to the type of printer managed by its printer driver. The `PrStlDialog` and `PrJobDialog` functions display the style and job dialog boxes defined by the resource file of the current printer.

For example, the standard style and job dialog boxes for the LaserWriter printer driver are shown in Figure 9-3 on page 9-7 and Figure 9-5 on page 9-8, respectively. The standard dialog boxes provided by the StyleWriter printer driver are shown in Figure 9-2 on page 9-6 and Figure 9-4 on page 9-7. Each dialog box has options that the user can set. If you want to use the standard style or job dialog box provided by the printer driver for the current printer, call the `PrStlDialog` function or the `PrJobDialog` function.

You may wish to add some additional options to these dialog boxes so that the user can customize the printing process even further. For example, Figure 9-12 illustrates a print job dialog box with two additional checkboxes: Print Selection Only and Skip Blank Pages.

Figure 9-12 A print job dialog box with additional checkboxes



You must follow these guidelines if you alter the style or job dialog boxes:

- Add additional options below the standard ones in the dialog box and don't change the standard ones—that is, don't delete, rearrange, or add new items in the existing list.
- Don't count on an item retaining its current position on the screen or in the dialog item list.
- Don't use more than half the smallest screen height for your items. (The smallest screen height is the 9-inch Macintosh Classic screen.) Printer drivers are allowed to expand the items in the standard dialog boxes to fill the top half of a 9-inch screen.
- If you want to add a lot of items to the dialog boxes, be aware this may confuse users. You should consider having your own separate dialog box in addition to the existing style and job dialog boxes.

You can customize a style or job dialog box by undertaking the following steps:

- Use the `PrOpen` procedure to open the Printing Manager.
- Use the `PrStlInit` or `PrJobInit` function to initialize a `TPrDlg` record. This record, described on page 9-50, contains the information needed to set up the style or job dialog box.
- Define an initialization routine that appends items to the printer driver's style or job dialog box. Your initialization routine should
  - use the Dialog Manager procedure `AppendDITL` to add items to the dialog box whose `TPrDlg` record you initialized with `PrStlInit` or `PrJobInit`
  - install two functions in the `TPrDlg` record: one—in the `pFltrProc` field—for handling events (such as update events for background applications) that the Dialog Manager doesn't handle in a modal dialog box, and another—in the `pItemProc` field—for handling events in the items added to the dialog box (for example, when the user clicks a checkbox that your application adds)
  - return a pointer to the `TPrDlg` record
- Pass the address of your initialization routine to the `PrDlgMain` function to display the dialog box.
- Respond to the dialog box as appropriate.
- Use the `PrClose` procedure when you are finished using the Printing Manager.

The event filter function pointed to in the `pFltrProc` field of the `TPrDlg` record extends the Dialog Manager's ability to handle events. When your application displays the style or job dialog box, you can use an event filter function to handle events that the Dialog Manager doesn't handle for modal dialog boxes. The chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* describes how to write an event filter function for the Dialog Manager.

The routine you supply in the `pItemProc` field of the `TPrDlg` record should handle events in the items you add to the dialog box. Sometimes called a *dialog hook*, this routine typically responds to clicks in the radio buttons or checkboxes that you add to the dialog box.

Listing 9-7 shows an application-defined routine called `DoPrintDialog` that specifies its own initialization function, called `MyPrDialogAppend`.

Listing 9-7 Installing an initialization function to alter the print job dialog box

```
FUNCTION DoPrintDialog: OSErr;   {display print job dialog box}
BEGIN
   PrOpen;     {open the Printing Manager}
   gPrintRec := THPrint(NewHandle(sizeof(TPrint)));{create a TPrint record}
   PrintDefault(gPrintRec);   {use default values for the TPrint record}
   gPrJobDialogBox := PrJobInit(gPrintRec);  {get a pointer to the }
                                             { invisible job dialog box}
   {use PrDlgMain to display the altered job dialog box}
   IF (PrDlgMain(gPrintRec, @MyPrDialogAppend)) THEN
      MyPrintDoc;
   PrClose;    {close the Printing Manager}
END;
```

The application-defined routine `MyPrDialogAppend` is shown in Listing 9-8. It uses the Resource Manager function `GetResource` to get a handle to an item list (`'DITL'`) resource containing the two extra checkboxes shown in Figure 9-12 on page 9-35. Using the Dialog Manager procedure `AppendDITL`, `MyPrDialogAppend` appends the items in this item list resource to the print job dialog box. Then `MyPrDialogAppend` installs the application's event filter function for modal dialog boxes. Finally, `MyPrDialogAppend` installs it own routine, called `HandleMyAppendedItems`, to handle clicks in the two newly installed checkboxes.

Listing 9-8 Adding items to a print job dialog box

```
FUNCTION MyPrDialogAppend (hPrint: THPrint): TPPrDlg;
VAR
  MyAppendDITLH: Handle;
BEGIN
   IF gDITLAppended = FALSE THEN
   BEGIN
      {first, get item list resource containing checkboxes}
      MyAppendDITLH := GetResource('DITL', kPrintingCheckBoxes);
```

```
      {next, append this item list resource to job dialog box}
      AppendDITL(DialogPtr(gPrJobDialogBox), MyAppendDITLH,
                 appendDITLBottom);
      gDITLAppended := TRUE;
   END;
   gFltrItemProc := LongInt(gPrJobDialogBox^.pFltrProc);
   {put an event filter function (to handle events that Dialog }
   { Manager doesn't handle in modal dialog boxes) }
   { in the pFltrProc field of the TPrDlg record}
   gPrJobDialogBox^.pFltrProc := ProcPtr(@MyEventFilter);
   gPrItemProc := LongInt(gPrJobDialogBox^.pItemProc);
   {put a dialog hook to handle clicks in appended items }
   { in the pItemProc field of the TPrDlg record}
   gPrJobDialogBox^.pItemProc := ProcPtr(@HandleMyAppendedItems);
   MyPrDialogAppend := gPrJobDialogBox;
END;
```

See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information about item list resources, event filter functions for modal dialog boxes, and the `AppendDITL` procedure. See the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for information about the `GetResource` function.

Writing an Idle Procedure

The printer driver for the current printer periodically calls an idle procedure while it sends a document to the printer. The `TPrJob` record contained in the `TPrint` record contains a pointer to an idle procedure in the `pIdleProc` field. If this field contains the value `NIL`, then the printer driver uses the Printing Manager's default idle procedure. The default idle procedure checks for Command-period keyboard events and sets the `iPrAbort` error code if one occurs, so that your application can cancel the print job at the user's request. However, the default idle procedure does not display a print status dialog box. It is up to the printer driver or your application to display a print status dialog box.

Most printer drivers display their own status dialog boxes. However, your application can display its own status dialog box that reports the current status of the printing operation to the user. If it does, your status dialog box should allow the user to press Command-period to cancel the printing operation, and it may also provide a button allowing the user to cancel the printing operation. To handle update events in your status dialog box, Command-period keyboard events, and clicks in your Cancel button (if you provide one), you should provide your own idle procedure. (See Figure 9-9 on page 9-14 for an example of an application-defined status dialog box.)

Here are several guidelines you must follow when writing your own idle procedure.

- If you designate an idle procedure, you must set the `pIdleProc` field of the `TPrJob` record *after* presenting the style and job dialog boxes, validating the `TPrint` record, and initializing the fields in the `TPrint` record (because the routines that perform these operations may reset the `pIdleProc` field to `NIL`). The `TPrJob` record is described on page 9-47.
- You must install your idle procedure in the `TPrint` record before calling the `PrOpenDoc` function. Otherwise, some printer drivers do not give the idle procedure any time to run.
- Do not attempt any printing from within the idle procedure, because the Printing Manager is *not* reentrant.
- Do not reference global variables unless you set up your own A5 world (as described in *Inside Macintosh: Processes*).
- If you use a modal dialog box to display printing status information, you must call the Event Manager function `WaitNextEvent` (described in the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*) to capture mouse events or the Command-period keyboard event that signals that the user wants to cancel printing. Do not call the `WaitNextEvent` function unless you display a modal dialog box.
- So that your application doesn't draw into a printing port, don't call the QuickDraw `OpenPicture` function or the `DrawPicture` procedure from your idle procedure without changing the current graphics port.
- Upon entry to the idle procedure, you must save the printing graphics port, and you must restore it upon exit if you draw anything within the idle procedure. If you don't save and restore the printing graphics port, upon return the printer driver draws into the graphics port of your dialog box instead of its own printing graphics port. To save the printer's printing graphics port, call the `GetPort` procedure when entering the idle procedure. Before you exit, call the `SetPort` procedure to set the port back to the printer driver's printing graphics port. (The `GetPort` and `SetPort` procedures are described in the chapter "Basic QuickDraw" in this book.)
- If your idle procedure changes the resource chain, you should save the reference number of the printer driver's resource file by calling the `CurResFile` function at the beginning of your idle procedure. (Any routine that changes the value of the global variable `TopMapHdl`, such as the `OpenResFile` function or the `UseResFile` procedure, changes the resource chain. Some printer drivers assume the resource chain does not change, and you may get an error if you change it.) When you exit from the idle procedure, restore the resource chain using the `UseResFile` procedure. If you are not changing the resource chain, you do not need to save the resource chain. (The `CurResFile`, `OpenResFile`, and `UseResFile` routines are described in the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*.)
- Avoid calling the `PrError` function within the idle procedure. Errors that occur while it is executing are usually temporary and serve only as internal flags for communication within the printer driver, not for the application. If you absolutely must call `PrError` within your idle procedure and an error occurs, do not cancel printing. Wait until the last called printing procedure returns and then check to see if the error still remains.

Listing 9-9 shows an application-defined idle procedure.

Listing 9-9 An idle procedure

```
PROCEDURE MyDoPrintIdle;
VAR
   oldPort:          GrafPtr;
   cursorRgn:        RgnHandle;
   event:            EventRecord;
   gotEvent:         Boolean;
   itemHIt:          Integer;
   handled, canceled:   Boolean;
BEGIN
   GetPort(oldPort);
   SetPort(gPrintStatusDlg);
   cursorRgn := NIL;
   gotEvent := WaitNextEvent(everyEvent, event, 15, cursorRgn);
   IF gotEvent THEN
   BEGIN
      handled := MyStatusHandleEvent(gPrintStatusDlg, event,
                                     itemHit);
      canceled := MyUserDidCancel;
      IF canceled THEN
         itemHit := kStopButton;
      handled := MyDoHandleHitsInStatusDBox(itemHit);
   END;
   MyUpdateStatusInformation(canceled);   {update status }
                               { information in dialog box}
   SetPort(oldPort);
END;
```

The application displays a modal dialog box for its status dialog box, and then installs `MyDoPrintIdle`, which calls the Event Manager function `WaitNextEvent` to capture events while the modal dialog box is displayed.

The `MyDoPrintIdle` procedure first saves the current graphics port (so that it can later restore it) and then sets the current port to the graphics port of the status dialog box. Your idle procedure should save, set, and restore the graphics port in this manner to avoid accidentally drawing in the printing graphics port.

The MyDoPrintIdle procedure then calls WaitNextEvent to get the current event and then calls its own routine to handle the event. (For example, the MyStatusHandleEvent function handles update and activate events.) By calling WaitNextEvent, MyDoPrintIdle gives background applications a chance to handle update events in their windows while the application in this example displays a modal dialog box. (The Dialog Manager does not give background applications a chance to handle update events in their windows when a modal dialog box is displayed.)

MyDoPrintIdle then calls another application-defined procedure to determine whether the user wishes to cancel the printing operation. The MyUserDidCancel function scans the event queue for keyboard events and mouse events. If it finds a Command-period keyboard event, it returns TRUE. The MyUserDidCancel function also returns TRUE if it finds mouse events indicating that the user clicked the Stop Printing button (that is, it uses the Dialog Manager function FindDialogItem to determine whether the mouse location specified in a mouse event is in the Stop Printing button). If the user clicks the Stop Printing button, MyUserDidCancel highlights the button appropriately.

To handle hits in the status dialog box, the MyDoHandleHitsInStatusDBox function simply checks the item number passed to it. For the Stop Printing button, MyDoHandleHitsInStatusDBox calls PrSetError, specifying the error code iPrAbort. For all other items, MyDoHandleHitsInStatusDBox sets the cursor to a spinning wristwatch cursor.

Finally, the MyDoPrintIdle procedure updates the items in the status dialog box that report status to the user.

Handling Printing Errors

You should always check for error conditions while printing by calling the PrError function. Errors returned may include AppleTalk and Operating System errors in addition to Printing Manager errors.

Note
Don't call PrError from within your idle procedure. See "Writing an Idle Procedure" on page 9-38 for more information. ◆

If you determine that an error has occurred after the completion of a printing routine, stop printing. Call the close routine that matches any open routine you have called. For example, if you call PrOpenDoc and receive an error, skip to the next call to PrCloseDoc; if you call PrOpenPage and get an error, skip to the next calls to PrClosePage and PrCloseDoc. Remember that, if you have called some open routine, you must call the corresponding close routine to ensure that the printer driver closes properly and that all temporary memory allocations are released and returned to the heap.

If you are using the `PrError` function and the `PrGeneral` procedure (described in "Getting and Setting Printer Information" beginning on page 9-28), be prepared to receive the following errors: `noSuchRsl`, `opNotImpl`, and `resNotFound`. In all three cases, your application should be prepared to continue to print without using the features of that particular opcode.

The `noSuchRsl` error means that the currently selected printer does not support the requested resolution. The `opNotImpl` error means that the currently selected printer does not support the particular `PrGeneral` opcode that you selected. The `resNotFound` error means the current printer driver does not support the `PrGeneral` procedure at all. This lack of support should not be a problem for your application, but you need to be prepared to deal with this error. If you receive a `resNotFound` result code from `PrError`, clear the error with a call to `PrSetError` with a value of `noErr` as the parameter; otherwise, `PrError` might still contain this error the next time you check it, which would prevent your application from printing.

Do not display any alert or dialog boxes to report an error until the end of the printing loop. Once at the end, check for the error again; if there is no error, assume that the printing completed normally. If the error is still present, then you can alert the user. This technique is important for two reasons.

- First, if you display a dialog box in the middle of the printing loop, it could cause errors that might terminate an otherwise normal printing operation. For example, if the printer is connected to an AppleTalk network, the connection might be terminated abnormally because the printer driver would be unable to respond to AppleTalk requests received from the printer while the dialog box was waiting for input from the user. If the printer does not hear from the Macintosh Operating System within a short period of time (anywhere from 30 seconds to 2 minutes, depending on the driver), it assumes that the Macintosh computer is no longer connected to the printer and times out. The timeout results in a prematurely broken connection, causing another error, to which the application must respond.
- Second, the printer driver may have already displayed its own dialog box in response to an error. In this instance, the printer driver posts an error to let the application know that something went wrong and it should cancel printing. For example, when a LaserWriter printer driver detects that the user has canceled printing, the driver posts an error to let the application know that it needs to cancel printing. Because the driver has already taken care of the error by displaying a dialog box, the error is reset to 0 before the printing loop is complete. The application should check for the error again at the end of the printing loop, and, if appropriate, the application can then display a dialog box.

Printing Manager Reference

This section describes the data structures and routines defined by the Printing Manager. When you print a document using the Printing Manager, the Printing Manager uses a printer driver to do the actual printing. A printer driver does any necessary translation of QuickDraw drawing routines and—when requested by your application—sends the translated instructions and data to the printer. It is the printer driver for the current printer that actually implements the routines defined by the Printing Manager. Every Printing Manager routine you call determines the current printer from a resource in the System file and then dispatches your call to the printer driver for that printer.

"Data Structures" shows the Pascal data structures defined by the Printing Manager. "Printing Manager Routines" describes the routines you can use to open and close the Printing Manager, display a print dialog box, print a document, and handle printing errors. "Application-Defined Routines" describes how you can provide your own idle procedure that handles events in a dialog box reporting the status of the print job, and how you can provide an initialization function that appends items to a print dialog box.

IMPORTANT

The burden of maintaining backward compatibility with early Apple printer models—as well as maintaining compatibility with over a hundred existing printer drivers—requires extra care on your part. When the Printing Manager was initially designed, it was intended to support ImageWriter printers directly attached to Macintosh computers with only a single floppy disk and 128 KB of RAM. Later, the Printing Manager was implemented on PostScript LaserWriter printer drivers for more powerful Macintosh computers sharing LaserWriter printers on networks. Since then, the Printing Manager has been implemented on a substantial—and unanticipated—number of additional Apple and third-party printer drivers, each in its own, slightly unique way. When you use Printing Manager routines and data structures, you should be especially wary of and defensive about possible error conditions. Because Apple has little control over the manner in which third parties support the Printing Manager in their printer drivers, you should test your application's printing code on as many printers as possible. ▲

Data Structures

This section shows the Pascal data structures defined by the Printing Manager.

You must create or ensure a valid `TPrint` record for every document before you can print it. This record specifies printer characteristics and the characteristics of a particular print job.

Contained in every `TPrint` record is a `TPrInfo` record, which specifies the vertical and horizontal resolutions, of the current printer and describes the page rectangle. A `TPrJob` record, which contains information about a particular print job—such as the range of pages to print, the number of copies, and a pointer to an idle procedure—is also contained in a `TPrint` record. A `TPrStl` record, which is also contained in a `TPrint` record, contains the device number of the current printer and the feed type to be used when printing the document.

The `PrPicFile` procedure returns printing status information in a record of data type `TPrStatus`. (You call the `PrPicFile` procedure for a printer using deferred printing.)

The `TPrDlg` record contains information necessary when altering the default style or job dialog box.

The `TPrPort` record describes a printing graphics port—the environment into which your application draws in order to print.

You use the `TGnlData`, `TGetRslBlk`, `TSetRslBlk`, `TDftBitsBlk`, and `TGetRotnBlk` records in conjunction with the `PrGeneral` procedure.

In almost all cases, printer drivers use the reserved fields in these data structures for device-dependent information. You should not rely on the availability or accuracy of this information when printing from your application.

TPrint

You must supply a record of data type `TPrint` for a document before it can be printed. (See “Creating and Using a TPrint Record” beginning on page 9-17 for information about how to supply a `TPrint` record for a document.)

In addition to other fields, the `TPrint` record includes three fields (`prInfo`, `prStl`, and `prJob`) that are defined by the `TPrInfo` record (described on page 9-46), the `TPrStl` record (described on page 9-48), and the `TPrJob` record (described on page 9-47). The `TPrint` record and the records within it contain information such as that needed by your application for printing a document.

```
TYPE
   TPPrint = ^TPrint;      {pointer to a TPrint record}
   THPrint = ^TPPrint;     {handle to a TPrint record}
   TPrint =
   RECORD
      iPrVersion: Integer; {reserved}
      prInfo:     TPrInfo; {resolution of device & page rectangle}
      rPaper:     Rect;    {paper rectangle}
      prStl:      TPrStl;  {printer driver number & feed type}
      prInfoPT:   TPrInfo; {reserved}
      prXInfo:    TPrXInfo;{reserved}
      prJob:      TPrJob;  {printing information from the job }
                           { dialog box}
      printX:     ARRAY[1..19] OF Integer;
   END;
```

Field descriptions

`iPrVersion` Reserved. To determine the version of the printer driver that initialized this `TPrint` record, use the `PrDrvrVers` function, which is described on page 9-79.

`prInfo` The information needed for page composition, contained in a `TPrInfo` record. See page 9-46 for a description of this record.

`rPaper` The paper rectangle. This rectangle encompasses the page rectangle, which is specified by the `rPage` field of the `TPrInfo` record.

`prStl` The printer's device number and the feed type, contained in a `TPrStl` record. See page 9-48 for a description of this record.

`prInfoPT` Reserved.

`prXInfo` Reserved.

`prJob` Information about this particular print job, contained in a `TPrJob` record. You use the `PrJobDialog` function to display the job dialog box. After the user closes the job dialog box, the `PrJobDialog` function updates the fields of the `TPrJob` record according to the user's choices. See page 9-47 for a description of this record.

`printX` Reserved.

If you try to use a `TPrint` record that's invalid for the current version of the Printing Manager or for the current printer, the printer driver corrects the record by setting its fields to default values.

Your application should not directly change the user-supplied data in the `TPrint` record; your application should use the `PrStlDialog` function and the `PrJobDialog` function (described on page 9-61 and page 9-62, respectively) or the `PrDlgMain` function (described on page 9-63) to allow the user to specify printing options, which the printer driver then translates to the appropriate fields in the `TPrint` record. The only fields you may need to set directly are those containing optional information in the `TPrJob` record (for example, the `pIdleProc` field, which contains a pointer to an idle procedure). Attempting to set other values directly in the `TPrint` record can produce unexpected results.

TPrInfo

The record defined by the data type `TPrInfo` contains printer information. The `prInfo` field of the `TPrint` record (described in the preceding section) contains a `TPrInfo` record, which in turn contains the vertical and horizontal resolutions of the printer and the coordinates of the page rectangle.

```
TYPE TPrInfo =          {printer information record}
   RECORD
      iDev:    Integer; {reserved}
      iVRes:   Integer; {vertical resolution of printer, in dpi}
      iHRes:   Integer; {horizontal resolution of printer, in dpi}
      rPage:   Rect;    {the page rectangle}
   END;
```

Field descriptions

`iDev` Reserved.

`iVRes` The printer's vertical resolution in dots per inch. The default value is 72, unless you have previously set the value for this record by calling the `PrGeneral` procedure with the `setRslOp` opcode (described in "Determining and Setting the Resolution of the Current Printer" on page 9-30).

`iHRes` The printer's horizontal resolution in dots per inch. The default value is 72, unless you have previously set the value for this record by calling the `PrGeneral` procedure with the `setRslOp` opcode.

`rPage` The page rectangle. As illustrated in Figure 9-6 on page 9-10, this rectangle is inside the paper rectangle, which is specified by the `rPaper` field of the `TPrint` record, described on page 9-44. You use the `PrStlDialog` function (described on page 9-61) to display the style dialog box. After the user closes the style dialog box, the `PrStlDialog` function updates the `rPage` field according to the user's choices.

TPrJob

The record defined by the data type `TPrJob` contains information about the print job. The `prJob` field of the `TPrint` record (described on page 9-44) contains a `TPrJob` record. You can set the contents of this record as a result of calling the `PrJobDialog` function (described on page 9-62) or the `PrJobInit` function (described on page 9-65), or by calling the `PrintDefault` procedure or `PrValidate` function (described on page 9-59 and page 9-60, respectively).

```
TYPE TPrJob =                  {print job record}
   RECORD
      iFstPage:   Integer;     {first page of page range}
      iLstPage:   Integer;     {last page of page range}
      iCopies:    Integer;     {number of copies}
      bJDocLoop:  SignedByte; {printing method: draft or deferred}
      fFromUsr:   Boolean;     {reserved}
      pIdleProc:  PrIdleProcPtr;
                               {pointer to an idle procedure}
      pFileName:  StringPtr;   {spool filename: NIL for default}
      iFileVol:   Integer;     {spool file volume; set to 0 }
                               { initially}
      bFileVers:  SignedByte; {spool file version; set to 0 }
                               { initially}
      bJobX:      SignedByte; {reserved}
END;
```

Field descriptions

`iFstPage` The page number of the first page to print.

`iLstPage` The page number of the last page to print.

`iCopies` The number of copies requested, which is also the number of times your application should send the document to the printer. However, some PostScript printer drivers handle multiple copies internally and set this value to 1.

`bJDocLoop` The printing method, as indicated by one of these constants:

```
CONST bDraftLoop = 0;   {draft-quality printing}
      bSpoolLoop = 1;   {deferred printing}
```

See the description of the `PrPicFile` procedure on page 9-71 on how to send a print job to the printer when this field contains the `bSpoolLoop` constant.

`fFromUsr` Reserved.

pIdleProc	A pointer to the idle procedure (described in "Writing an Idle Procedure" on page 9-38) for this printing operation. A value of NIL specifies the default idle procedure.
pFileName	The name of the spool file (normally "Print File") for deferred printing. This field is maintained by the printer driver, and your application should not change or rely on its value.
iFileVol	The volume reference number of the spool file. This field is maintained by the printer driver, and your application should not change or rely on its value.
bFileVers	The version number of the spool file, initialized to 0.
bJobX	Reserved.

TPrStl

The prStl field of the TPrint record (described on page 9-44) contains a TPrStl record, which in turn contains the device number of the current printer and the feed type currently selected (either paper cassette or manual). All other fields are reserved.

```
TYPE TPrStl =              {printing style record}
   RECORD
      wDev:    Integer;    {device number of printer}
      iPageV:  Integer;    {reserved}
      iPageH:  Integer;    {reserved}
      bPort:   SignedByte; {reserved}
      feed:    TFeed;      {feed type}
   END;
```

Field descriptions

wDev	The device number of the current printer (in the high-order byte of this field). The low-order byte of this field is reserved.
iPageV	Reserved.
iPageH	Reserved.
bPort	Reserved.

feed	The feed type currently selected. The possible values are defined by the TFeed data type:
	
	`TYPE TFeed =`
	`(feedCut,feedFanfold,feedMechCut,feedOther);`

TPrStatus

The `PrPicFile` procedure (described on page 9-71) returns printing status information in a record of data type `TPrStatus`. (You call the `PrPicFile` procedure for a printer using deferred printing.)

```
TYPE TPrStatus =              {printing status record}
   RECORD
      iTotPages:  Integer;    {total pages in print file}
      iCurPage:   Integer;    {current page number}
      iTotCopies: Integer;    {total copies requested}
      iCurCopy:   Integer;    {current copy number}
      iTotBands:  Integer;    {reserved}
      iCurBand:   Integer;    {reserved}
      fPgDirty:   Boolean;    {TRUE if current page has been }
                              { written to}
      fImaging:   Boolean;    {reserved}
      hPrint:     THPrint;    {handle to the active TPrint record}
      pPrPort:    TPPrPort;   {pointer to the active printing }
                              { port}
      hPic:       PicHandle;  {handle to the active picture}
   END;
```

Field descriptions

`iTotPages` The total number of pages being printed. This is the value of the `iLstPage` field minus the value of the `iFstPage` field, which are both in the `TPrJob` record (described on page 9-47).

`iCurPage` The sequence number of the page currently being printed. For example, if the user prints pages 10 through 15 of a 20-page document, the value of the `iCurPage` field for page 10 is 1.

`iTotCopies` The total number of copies requested. This value may be different from the value of the `iCopies` field in the `TPrJob` record.

`iCurCopy` The number of the current copy being printed.

`iTotBands` Reserved.

`iCurBand` Reserved.

`fPgDirty` A flag indicating whether the printer has begun printing the current page. Set to `TRUE` if there has been any imaging on the current page.

`fImaging` A flag indicating whether the printer driver is in the middle of an imaging call.

`hPrint` A handle to the current `TPrint` record (described on page 9-44).

`pPrPort` A pointer to the `TPrPort` record for the current printing graphics port (described on page 9-51).

`hPic` A handle to the active picture. This is used by the printer driver; your application should not alter it.

TPrDlg

The `TPrDlg` record contains information necessary when altering the default style or job dialog box for the current printer driver. The `PrStlInit` function (described on page 9-64) returns a `TPrDlg` record with information for a style dialog box; the `PrJobInit` function (described on page 9-65) returns a `TPrDlg` record with information for a job dialog box.

```
TYPE
TPPrDlg  = TPrDlg;
TPrDlg   =                          {print dialog box record}
   RECORD
      Dlg:        DialogRecord;     {a dialog record}
      pFltrProc:                    {pointer to event filter}
                  ModalFilterProcPtr;
      pItemProc:  PItemProcPtr;     {pointer to item-handling }
                                    { procedure}
      hPrintUsr:  THPrint;          {handle to a TPrint record}
      fDoIt:      Boolean;          {TRUE means user clicked OK}
      fDone:      Boolean;          {TRUE means user clicked }
                                    { OK or Cancel}
      lUser1:     LongInt;          {storage for your application}
      lUser2:     LongInt;          {storage for your application}
      lUser3:     LongInt;          {storage for your application}
      lUser4:     LongInt;          {storage for your application}
   END;
```

Field descriptions

`Dlg` A dialog record that represents either the style or job dialog box. This record is described in the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

`pFltrProc` A pointer to an event filter function that handles events the Dialog Manager does not respond to (such as disk-inserted events and update events for background applications) in a modal dialog box. Event filter functions for modal dialog boxes are described in the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

`pItemProc` A pointer to a routine (sometimes called a *dialog hook*) that responds to events in those items—such as checkboxes and radio buttons—that your application has added to the dialog box. See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information about responding to events in dialog boxes.

hPrintUsr	A handle to a TPrint record (described on page 9-44) for a document.
fDoIt	A Boolean value indicating whether the user has confirmed the dialog box. A value of TRUE means the user has confirmed it by clicking the OK button.
fDone	A Boolean value indicating whether the user's interaction is completed. A value of TRUE means the user has clicked either the OK or Cancel button.
lUser1	In this field and the following fields, your application can store any kind of data you wish for the dialog box.
lUser2	Available for your application's use.
lUser3	Available for your application's use.
lUser4	Available for your application's use.

Figure 9-3 on page 9-7 shows a style dialog box. Figure 9-5 on page 9-8 shows a job dialog box. You can find information on how to customize a dialog box in "Altering the Style or Job Dialog Box" beginning on page 9-35.

TPrPort

The record of the data type TPrPort contains a graphics port and a pointer to a QDProcs record.

```
TYPE TPrPort =                  {printing graphics port}
   RECORD
      gPort:      GrafPort;     {graphics port for printing}
      gProcs:     QDProcs;      {procedures for printing }
                                {  in the graphics port}
      lGParam1:   LongInt;      {reserved}
      lGParam2:   LongInt;      {reserved}
      lGParam3:   LongInt;      {reserved}
      lGParam4:   LongInt;      {reserved}
      fOurPtr:    Boolean;      {reserved}
      fOurBits:   Boolean;      {reserved}
END;
```

Field descriptions

gPort	Either a CGrafPort or GrafPort record, depending on whether the current printer supports color or grayscale and depending on whether Color QuickDraw is available. If you need to determine the type of graphics port, you can check the high bit in the rowBytes field of the record contained in the gPort field; if this bit is set, the printing graphics port is based on a CGrafPort record.

gProcs	A QDProcs record that contains pointers to routines that the printer driver may have designated to take the place of QuickDraw routines. See the chapter "Basic QuickDraw" in this book for more information about the QDProcs record.
lGParam1	Reserved.
lGParam2	Reserved.
lGParam3	Reserved.
lGParam4	Reserved.
fOurPtr	Reserved.
fOurBits	Reserved.

TGnlData

The record of data type TGnlData is the basic record used by the PrGeneral procedure (described beginning on page 9-72). Although no opcode of PrGeneral uses the TGnlData record, all other records created for PrGeneral are based on this record.

```
TYPE TGnlData =
   RECORD
      iOpCode:    Integer; {opcode passed to PrGeneral}
      iError:     Integer; {result code returned by PrGeneral}
      lReserved:  LongInt; {more fields here depending on opcode}
END;
```

Field descriptions

iOpCode — The opcode that is passed to PrGeneral to obtain the requested feature. There are five possible opcodes; you can use the following constants or the values they represent to specify one of these opcodes

```
CONST {opcodes used with PrGeneral}
 getRslDataOp  = 4;  {get resolutions for the }
                     { current printer}
 setRslOp      = 5;  {set resolutions for a }
                     { TPrint record}
 draftBitsOp   = 6;  {force enhanced draft- }
                     { quality printing}
 noDraftBitsOp = 7;  {cancel enhanced draft- }
                     { quality printing}
 getRotnOp     = 8;  {get page orientation of }
                     { a TPrint record}
```

iError — The result code returned by PrGeneral.

`lReserved` Reserved. Additional fields may follow this field, depending on the opcode used. See the descriptions of the `TGetRslBlk` (in the next section), `TSetRslBlk` (on page 9-54), `TDftBitsBlk` (on page 9-55), and `TGetRotnBlk` (on page 9-56) records.

TGetRslBlk

You pass a record defined by the data type `TGetRslBlk` to the `PrGeneral` procedure when you use the `getRslDataOp` opcode. When the `PrGeneral` procedure completes, the `TGetRslBlk` record contains the resolutions available on the current printing device. For information on how to use the `TGetRslBlk` record with the `PrGeneral` procedure, see "Determining and Setting the Resolution of the Current Printer" on page 9-30.

```
TYPE TGetRslBlk =          {get-resolution record}
   RECORD
      iOpCode:    Integer; {the getRslDataOp opcode}
      iError:     Integer; {result code returned by PrGeneral}
      lReserved:  LongInt; {reserved}
      iRgType:    Integer; {printer driver version number}
      xRslRg:     TRslRg;  {x-direction resolution range}
      yRslRg:     TRslRg;  {y-direction resolution range}
      iRslRecCnt: Integer; {number of resolution records}
      rgRslRec:            {array of resolution records}
                  ARRAY[1..27] OF TRslRec;
   END;
```

Field descriptions

`iOpCode` The opcode `getRslDataOp`.

`iError` The result code returned by `PrGeneral`.

`lReserved` Reserved.

`iRgType` The version number returned by the printer driver.

`xRslRg` The resolution range supported for the x direction. This field contains a record defined by the data type `TRslRg`:

```
TYPE TRslRg =
   RECORD
      iMin: Integer; {minimum resolution supported}
      iMax: Integer; {maximum resolution supported}
   END;
```

If the current printer does not support variable resolution, the values in the `iMin` and `iMax` fields are 0.

`yRslRg` The resolution range supported for the y direction. This field contains a record defined by the data type `TRslRg`, which is shown in the preceding description for the `xRslRg` field. If the current printer does not support variable resolution, the values in the `iMin` and `iMax` fields are 0.

`iRslRecCnt` The number of `TRslRec` records used by a particular printer driver (up to 27) if it supports discrete resolution. If it supports variable resolution, this field contains 0. The `TRslRec` record is described next.

`rgRslRec` An array of records defined by the `TRslRec` data type, each specifying a discrete resolution at which the current printer can print an image. A printer driver may contain up to 27 separate `TRslRec` records.

```
TYPE TRslRec =
   RECORD
      iXRsl: Integer;   {discrete resolution, }
                        { x direction}
      iYRsl: Integer;   {discrete resolution, }
                        { y direction}
   END;
```

TSetRslBlk

You pass a record defined by the data type `TSetRslBlk` to the `PrGeneral` procedure when you use the `setRslOp` opcode. You use this record to specify the resolutions that you want to use when printing the data associated with a `TPrint` record. For information on how to use the `TSetRslBlk` record with the `PrGeneral` procedure, see "Determining and Setting the Resolution of the Current Printer" beginning on page 9-30.

```
TYPE TSetRslBlk =          {set-resolution record}
   RECORD
      iOpCode:    Integer; {the setRslOp opcode}
      iError:     Integer; {result code returned by PrGeneral}
      lReserved:  LongInt; {reserved}
      hPrint:     THPrint; {handle to the current TPrint record}
      iXRsl:      Integer; {x-direction resolution you want}
      iYRsl:      Integer; {y-direction resolution you want}
   END;
```

Field descriptions

`iOpCode` The opcode `setRslOp`.

`iError` The result code returned by `PrGeneral`.

`lReserved` Reserved.

hPrint	A handle to a TPrint record, which is described on page 9-44. Your application should have already created this TPrint record and passed it to the PrintDefault or PrValidate routine to make sure that all of the information in the TPrint record is valid.
iXRsl	The resolution in the x direction that you want the printer to use when printing the data associated with the TPrint record specified in the hPrint field.
iYRsl	The resolution in the y direction that you want the printer to use when printing the data associated with the TPrint record specified in the hPrint field.

After calling PrGeneral with the setRslOp opcode, you can determine whether the request was successful by examining the iError field of the TSetRslBlk record. If the iError field returns noErr, the Printing Manager updated the TPrint record with the specified resolution, which the printer uses when printing the data associated with this TPrint record. If the iError field returns noSuchRsl, the current printer doesn't support the requested resolution, and the printer driver does not change the setting in the TPrint record.

TDftBitsBlk

You pass a record defined by the data type TDftBitsBlk to the PrGeneral procedure when you use the draftBitsOp or noDraftBitsOp opcode. For information on how to use the TDftBitsBlk record with the PrGeneral procedure, see "Enhancing Draft-Quality Printing" on page 9-33.

```
TYPE TDftBitsBlk =          {draft bits record}
   RECORD
      iOpCode:    Integer; {draftBitsOp or noDraftBitsOp opcode}
      iError:     Integer; {result code returned by PrGeneral}
      lReserved:  LongInt; {reserved}
      hPrint:     THPrint; {handle to the current TPrint record}
   END;
```

Field descriptions

iOpCode	Either the draftBitsOp or noDraftBitsOp opcode.
iError	The result code returned by the PrGeneral procedure.
lReserved	Reserved.
hPrint	A handle to a TPrint record, which is described on page 9-44. Your application should have already created this TPrint record and passed it to the PrintDefault or PrValidate routine to make sure that all of the information in the TPrint record is valid. The PrintDefault and PrValidate routines are described on page 9-59 and page 9-60, respectively.

TGetRotnBlk

You pass a record defined by the data type `TGetRotnBlk` to the `PrGeneral` procedure when you use the `getRotnOp` opcode. `PrGeneral` returns it with a Boolean variable that tells you whether the user has selected landscape orientation. For information on how to use the `TGetRotnBlk` record with the `PrGeneral` procedure, see "Determining Page Orientation" on page 9-32.

```
TYPE TGetRotnBlk =            {page orientation record}
   RECORD
      iOpCode:    Integer;    {the getRotnOp opcode}
      iError:     Integer;    {result code returned by PrGeneral}
      lReserved:  LongInt;    {reserved}
      hPrint:     THPrint;    {handle to current TPrint record}
      fLandscape: Boolean;    {TRUE if user selected landscape }
                              { printing}
      bXtra:      SignedByte; {reserved}
   END;
```

Field descriptions

`iOpCode`	The opcode `getRotnOp`.
`iError`	The result code returned by the `PrGeneral` procedure.
`lReserved`	Reserved.
`hPrint`	A handle to a `TPrint` record, which is described on page 9-44. Your application should have already created this `TPrint` record and passed it through the `PrintDefault` or `PrValidate` routine to make sure that all of the information in the `TPrint` record is valid. The `PrintDefault` and `PrValidate` routines are described on page 9-59 and page 9-60, respectively.
`fLandscape`	A Boolean value that determines whether the user has selected landscape orientation in the style dialog box. A value of `TRUE` indicates the user has selected landscape orientation.
`bXtra`	Reserved.

Printing Manager Routines

This section describes the routines you use to open and close the current printer driver, produce or alter a style or job dialog box, print a document, and handle printing errors.

Opening and Closing the Printing Manager

You must always use the `PrOpen` procedure to open the current printer driver before attempting to print, and you must use the `PrClose` procedure to close the current printer driver when printing is finished.

PrOpen

Use the `PrOpen` procedure to prepare the current printer driver for use.

```
PROCEDURE PrOpen;
```

DESCRIPTION

The `PrOpen` procedure opens the Printing Manager and the current printer driver.

SPECIAL CONSIDERATIONS

You must always use the `PrOpen` procedure before using any other Printing Manager routines, and you must balance every call to `PrOpen` with a call to `PrClose`, which is described in the next section.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PrOpen` procedure are

Trap macro	Selector
`_PrGlue`	$C8000000

SEE ALSO

For an example of the use of `PrOpen`, see Listing 9-2 on page 9-20.

PrClose

When you are finished using Printing Manager routines, use the `PrClose` procedure to close the Printing Manager and release the memory it occupies.

```
PROCEDURE PrClose;
```

DESCRIPTION

The `PrClose` procedure is the call that balances a call to the `PrOpen` procedure.

SPECIAL CONSIDERATIONS

If you have opened the printer driver with the `PrOpen` procedure, do not call the `PrDrvrClose` procedure (described on page 9-80) to close it. Similarly, do not close the printer driver with `PrClose` if you opened it with the `PrDrvrOpen` procedure (described on page 9-79).

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PrClose` procedure are

Trap macro	Selector
`_PrGlue`	$D0000000

SEE ALSO

For an example of the use of `PrClose`, see Listing 9-2 beginning on page 9-20.

Initializing and Validating TPrint Records

You must set the fields of the `TPrint` record to the values for the current printer driver or, if a `TPrint` record already exists, you must verify that the information in the `TPrint` record is correct. The `PrintDefault` procedure fills in a `TPrint` record with the default values for the current printer.

If the `TPrint` record is not valid for the current printer driver, the document does not print. The `PrValidate` function ensures that the `TPrint` record is compatible with the current version of the printer driver for the current printer. These functions may change the coordinates of the page rectangle or any other value in the `TPrint` record; you should not assume any values will remain the same.

PrintDefault

When you create a `TPrint` record, you use the `PrintDefault` procedure to initialize the fields of the `TPrint` record according to the current printer's default values for resolution, number of copies, and so on.

```
PROCEDURE PrintDefault (hPrint: THPrint);
```

`hPrint` A handle to a `TPrint` record (described on page 9-44), which may be a new record or an existing one from a document.

DESCRIPTION

The default values for the current printer are stored in the printer driver's resource file. The `PrintDefault` procedure puts these values in the `TPrint` record, replacing the ones that may already be there. The `PrintDefault` procedure calls the `PrValidate` function (described in the next section) to ensure that the `TPrint` record is compatible with the current version of the printer driver.

SPECIAL CONSIDERATIONS

You should never call `PrintDefault` between the pages of a document.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PrintDefault` procedure are

Trap macro	Selector
`_PrGlue`	$20040480

SEE ALSO

See "The TPrint Record and the Printing Loop" on page 9-11 and see page 9-44 for information on the `TPrint` record. For an example of the use of `PrintDefault`, see Listing 9-7 on page 9-37.

PrValidate

When you have a `TPrint` record, whether an existing one from the current document or a new one you have just created, you can use the `PrValidate` function to ensure that the contents of the specified `TPrint` record are compatible with the current version of the printer driver for the current printer.

```
FUNCTION PrValidate (hPrint: THPrint): Boolean;
```

`hPrint` A handle to a `TPrint` record, which may be a new record or an existing one from a document.

DESCRIPTION

If the `TPrint` record is valid, the `PrValidate` function returns `FALSE`, meaning there is no change. If the record is invalid, the function returns `TRUE` and the Printing Manager adjusts the record with the default values stored in the printer resource file for the current printer.

The `PrValidate` function also makes sure that all the information in the `TPrint` record is internally self-consistent and updates the `TPrint` record as necessary. These changes do not affect the function's Boolean result.

If you have just created a `TPrint` record by using the `PrintDefault` procedure, you do not need to call `PrValidate`. The `PrintDefault` procedure does this automatically.

SPECIAL CONSIDERATIONS

You should never call `PrValidate` between the pages of a document. This restriction holds as well for the `PrStlDialog` and `PrJobDialog` functions (described on page 9-61 and page 9-62, respectively) and the `PrintDefault` procedure (described on page 9-59), which call `PrValidate`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PrValidate` function are

Trap macro	Selector
`_PrGlue`	$52040498

SEE ALSO

For examples of the use of `PrValidate`, see Listing 9-1 on page 9-17 and Listing 9-2 on page 9-20.

Displaying and Customizing the Print Dialog Boxes

The style and job dialog boxes allow the user to tell your application how to print the document: its page orientation, number of copies, page range, and so on. The `PrStlDialog` and `PrJobDialog` functions display the standard style and job dialog boxes as provided by the resource file of the current printer driver. The `PrDlgMain` function, along with the `PrStlInit` and `PrJobInit` functions, allows you to customize the current printer driver's style and job dialog boxes.

The `PrJobMerge` procedure allows you to use one job dialog box for several print jobs, such as when the user prints several documents from the Finder.

PrStlDialog

You can use the `PrStlDialog` function to display the style dialog box provided by the resource file for the current printer driver.

```
FUNCTION PrStlDialog (hPrint: THPrint): Boolean;
```

`hPrint` A handle to a `TPrint` record (described on page 9-44), which may be a new record or an existing one from a document.

DESCRIPTION

The `PrStlDialog` function gets the initial settings to display in the style dialog box from the `TPrint` record specified in the `hPrint` parameter. The user specifies the page dimensions and other information needed for page setup through the style dialog box. Your application should display this dialog box when the user chooses Page Setup from the File menu.

If the user confirms the dialog box, the `PrStlDialog` function returns `TRUE`. The `PrStlDialog` function saves the results of the dialog box in the specified `TPrint` record and calls the `PrValidate` function (described on page 9-60). Otherwise, the `TPrint` record is left unchanged and the function returns `FALSE`.

SPECIAL CONSIDERATIONS

You should never call `PrStlDialog` between the pages of a document.

You must call the `PrOpen` procedure (described on page 9-57) prior to calling `PrStlDialog`, and you must call the `PrClose` procedure (described on page 9-58) afterward, because the current printer driver must be open in order for your application to successfully call `PrStlDialog`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PrStlDialog` function are

Trap macro	Selector
`_PrGlue`	$2A040484

SEE ALSO

See Figure 9-3 on page 9-7 for an example of a style dialog box. For more information on the use of a style dialog box, see "Getting Printing Preferences From the User" beginning on page 9-5. For information on how to customize a style dialog box, see "Altering the Style or Job Dialog Box" beginning on page 9-35.

PrJobDialog

You can use the `PrJobDialog` function to display the job dialog box provided by the resource file for the current printer driver.

```
FUNCTION PrJobDialog (hPrint: THPrint): Boolean;
```

`hPrint` A handle to a `TPrint` record (described on page 9-44), which may be a new record or an existing one from a document.

DESCRIPTION

The `PrJobDialog` function gets the initial settings to display in the job dialog box from the `TPrint` record specified in the `hPrint` parameter. The user specifies the print quality, the range of pages to print, and other information in the job dialog box. Your application should display this dialog box when the user chooses Print from the File menu.

If the user confirms the dialog box, the `PrJobDialog` function updates both the `TPrint` record and the printer driver resource file and calls the `PrValidate` function, and the `PrJobDialog` function returns `TRUE`. Even if the function returns `FALSE`, the `PrJobDialog` function may have updated the `TPrint` record.

SPECIAL CONSIDERATIONS

You should proceed with the requested printing operation only if the `PrJobDialog` function returns `TRUE`. You should never call `PrJobDialog` between the pages of a document.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PrJobDialog` function are

Trap macro	Selector
`_PrGlue`	$32040488

SEE ALSO

See Figure 9-5 on page 9-8 for an example of a job dialog box. For more information on the use of a job dialog box, see "Getting Printing Preferences From the User" beginning on page 9-5. For information on how to customize a job dialog box, see "Altering the Style or Job Dialog Box" beginning on page 9-35.

PrDlgMain

To display a customized style or job dialog box for the current printer driver, use the `PrDlgMain` function.

```
FUNCTION PrDlgMain (hPrint: THPrint; pDlgInit: PDlgInitProcPtr):
                    Boolean;
```

`hPrint` A handle to a `TPrint` record (described on page 9-44), which may be a new record or an existing one from a document.

`pDlgInit` A pointer to your own initialization procedure or a pointer to one of the default initialization functions (`PrStlInit`, which is described in the next section, or `PrJobInit`, which is described on page 9-65).

DESCRIPTION

You use the `PrDlgMain` function to display a style or job dialog box that your application has altered. (If you use the standard style and job dialog boxes, you do not need to call `PrDlgMain`; instead, you can simply call the `PrStlDialog` or `PrJobDialog` function, described on page 9-61 and page 9-62, respectively.)

If you want to customize a style or job dialog box, first call `PrStlInit`, which is described in the next section, or `PrJobInit`, which is described on page 9-65, to get a pointer to the `TPrDlg` record (described on page 9-50) for that dialog box. The `PrStlInit` function returns a pointer to the `TPrDlg` record for the style dialog box of the current printer driver; the `PrJobInit` function returns a pointer to the `TPrDlg` record of the job dialog box for the current printer driver. You should supply the `TPrDlg` record for your customized dialog box with a function that handles events that the Dialog Manager doesn't handle, and with another function that handles events in the items you add to the dialog box.

When `PrDlgMain` returns `TRUE`, you should proceed with the requested printing operation.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PrDlgMain` function are

Trap macro	Selector
`_PrGlue`	$4A040894

SEE ALSO

For more information about customizing style or job dialog boxes, see "Altering the Style or Job Dialog Box" beginning on page 9-35.

PrStlInit

To initialize a `TPrDlg` record for a customized style dialog box, use the `PrStlInit` function.

```
FUNCTION PrStlInit (hPrint: THPrint): TPPrDlg;
```

`hPrint` A handle to a `TPrint` record (described on page 9-44), which may be a new record or an existing one from a document.

DESCRIPTION

The `PrStlInit` function returns a pointer to a `TPrDlg` record (described on page 9-50) for the style dialog box defined in the resource file for the current printer driver. As described in "Altering the Style or Job Dialog Box" beginning on page 9-35, you can then alter the dialog box by adding your own items. You must use the `PrDlgMain` function (described on page 9-63) to display the dialog box.

You need to use `PrStlInit` only if you are customizing the default style dialog box provided by the printer driver. To initialize and display the default style dialog box, use the `PrStlDialog` function, which is described on page 9-61.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PrStlInit` function are

Trap macro	Selector
`_PrGlue`	$3C04040C

PrJobInit

To initialize a `TPrDlg` record for a customized job dialog box, use the `PrJobInit` function.

```
FUNCTION PrJobInit (hPrint: THPrint): TPPrDlg;
```

`hPrint` A handle to a `TPrint` record (described on page 9-44), which may be a new record or an existing one from a document.

DESCRIPTION

The `PrJobInit` function returns a pointer to a `TPrDlg` record (described on page 9-50) for the job dialog box defined in the resource file for the current printer driver. As described in "Altering the Style or Job Dialog Box" beginning on page 9-35, you can then alter the dialog box by adding your own items. You must use the `PrDlgMain` function (described on page 9-63) to display the dialog box.

You need to use `PrJobInit` only if you are customizing the job dialog box provided by the printer driver. To initialize and display the default job dialog box, use the `PrJobDialog` function, which is described on page 9-62.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PrJobInit` function are

Trap macro	Selector
`_PrGlue`	$44040410

SEE ALSO

Listing 9-7 on page 9-37 illustrates how to use `PrJobInit` when customizing the job dialog box.

PrJobMerge

You can use the `PrJobMerge` procedure to apply the same information previously specified by the user through the job dialog box to several `TPrint` records. This is useful when the user prints from the Finder. The `PrJobMerge` procedure allows you to solicit information from the user just once and then use this information to print several documents.

```
PROCEDURE PrJobMerge (hPrintSrc: THPrint; hPrintDst: THPrint);
```

`hPrintSrc` A handle to a `TPrint` record (described on page 9-44) as previously returned by the `PrJobDialog` function (described on page 9-62).

`hPrintDst` A handle to a `TPrint` record for another document.

DESCRIPTION

The `PrJobMerge` procedure first calls the `PrValidate` function (described on page 9-60) for both `TPrint` records referenced by the `hPrintSrc` and `hPrintDst` parameters. It then copies all of the information previously set as a result of a job dialog box from the `TPrint` record in the `hPrintSrc` parameter to the `TPrint` record in the `hPrintDst` parameter while preserving the values set by the style dialog box for that `TPrint` record (for instance, landscape orientation). Finally, the `PrJobMerge` procedure makes sure that all the fields of the `TPrint` record named by the `hPrintDst` parameter are internally self-consistent. You must call `PrJobMerge` for each document the user wants to print.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PrJobMerge` procedure are

Trap macro	Selector
`_PrGlue`	$5804089C

Printing a Document

In addition to using the `PrOpen` and `PrClose` procedures (described on page 9-57 and page 9-58, respectively) to open and close the current printer driver, you must open a printing graphics port for a document and open each page of the document before printing the page. You must close each page after printing it, and you must close the printing graphics port after printing the last page of the document. The `PrOpenDoc` function and `PrCloseDoc` procedure open and close the printing graphics port for the document, and the `PrOpenPage` and `PrClosePage` procedures open and close the current page.

You must use the `PrPicFile` procedure to complete printing for a driver using deferred printing.

PrOpenDoc

Use the `PrOpenDoc` function to initialize a printing graphics port for use in printing a document.

```
FUNCTION PrOpenDoc (hPrint: THPrint; pPrPort: TPPrPort;
                    pIOBuf: Ptr): TPPrPort;
```

`hPrint` A handle to a `TPrint` record (described on page 9-44), which may be a new record or an existing one from a document. You should call the `PrintDefault` procedure (described on page 9-59) or the `PrValidate` function (described on page 9-60) for this `TPrint` record before calling `PrOpenDoc`.

`pPrPort` A pointer to a printing graphics port. If you set this parameter to `NIL`, `PrOpenDoc` allocates a new printing graphics port in the heap.

`pIOBuf` A pointer to an area of memory to be used as an input and output buffer. If you set this parameter to `NIL`, `PrOpenDoc` uses the volume buffer for the deferred spool file's volume. If you allocate your own buffer, it must be exactly 522 bytes.

DESCRIPTION

The `PrOpenDoc` function initializes and returns a pointer to a printing graphics port for use in printing a document. (The `TPrPort` record that defines a printing graphics port is described on page 9-51.) The `PrOpenDoc` function also sets the current graphics port to the printing graphics port.

Because both the printing graphics port and input and output buffer are nonrelocatable objects, you may want to allocate them yourself using the `pPrPort` and `pIOBuf` parameters (to avoid fragmenting the heap).

SPECIAL CONSIDERATIONS

You must balance a call to `PrOpenDoc` with a call to the `PrCloseDoc` procedure, which is described in the next section.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PrOpenDoc` function are

Trap macro	Selector
`_PrGlue`	$04000C00

SEE ALSO

For an example of the use of `PrOpenDoc`, see Listing 9-2 beginning on page 9-20. For a description of the `PrValidate` function and `PrintDefault` procedure, see page 9-60 and page 9-59, respectively.

PrCloseDoc

Use the `PrCloseDoc` procedure to close a printing graphics port previously opened with the `PrOpenDoc` procedure.

```
PROCEDURE PrCloseDoc (pPrPort: TPPrPort);
```

`pPrPort` A pointer to a printing graphics port. (The `TPrPort` record that defines a printing graphics port is described on page 9-51.)

DESCRIPTION

The `PrCloseDoc` procedure closes the current printing graphics port. You typically use `PrCloseDoc` after sending the last page of a document to the printer with the `PrClosePage` procedure (described on page 9-70).

When you use `PrCloseDoc` to close a printing graphics port, printer drivers respond in a manner appropriate for the printers they control. Many drivers, including the LaserWriter driver, start a print job after your application calls `PrCloseDoc`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PrCloseDoc` procedure are

Trap macro	Selector
`_PrGlue`	$08000484

SPECIAL CONSIDERATIONS

For deferred printing on an ImageWriter printer, call the `PrError` function (described on page 9-75) to find out whether spooling succeeded before using `PrCloseDoc`. If spooling succeeded, call the `PrPicFile` procedure (described on page 9-71).

SEE ALSO

For an example of the use of `PrCloseDoc`, see Listing 9-2 beginning on page 9-20.

PrOpenPage

Use the `PrOpenPage` procedure to begin to print a new page.

```
PROCEDURE PrOpenPage (pPrPort: TPPrPort; pPageFrame: TPRect);
```

`pPrPort` A pointer to a printing graphics port. (The `TPrPort` record that defines a printing graphics port is described on page 9-51.)

`pPageFrame`
For deferred printing, a pointer to a rectangle to be used as the QuickDraw picture frame for this page. To print the page with no scaling, specify `NIL` to use the rectangle in the `rPage` field of the `TPrInfo` record as the picture frame.

DESCRIPTION

The `PrOpenPage` procedure sets up the printing graphics port to print a new page. After calling `PrOpenPage`, your application should draw the data for that page and then call the `PrClosePage` procedure, which is described in the next section.

The page is printed only if it falls within the page range stored in the `TPrJob` record contained in the `TPrint` record supplied to the `PrOpenDoc` function (described on page 9-67).

If the user has chosen deferred printing for a printer driver that supports deferred printing, the driver uses the QuickDraw procedure `DrawPicture` to scale the rectangle named in the `pPageFrame` parameter so that it coincides with the rectangle specified in the `rPage` field of the `TPrInfo` record (which is contained in the `TPrint` record supplied to the `PrOpenDoc` function). Unless you want the printout to be scaled, you should set the `pPageFrame` parameter to `NIL`—this uses the rectangle in the `rPage` field as the picture frame, so that the page is printed with no scaling.

SPECIAL CONSIDERATIONS

You must balance every call to `PrOpenPage` with a call to `PrClosePage`.

The printing graphics port is completely reinitialized by `PrOpenPage`. Therefore, you must set graphics port features such as the font family and font size for every page that you draw after you call this procedure.

Don't call the QuickDraw function `OpenPicture` while a page is open (after a call to `PrOpenPage` but before calling `PrClosePage`). You can, however, call the `DrawPicture` procedure at any time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PrOpenPage` procedure are

Trap macro	Selector
`_PrGlue`	$10000808

SEE ALSO

For an example of the use of `PrOpenPage`, see Listing 9-2 beginning on page 9-20. The QuickDraw routines `OpenPicture` and `DrawPicture` are described in the chapter "Pictures" in this book.

PrClosePage

Use the `PrClosePage` procedure to finish the printing of the current page.

```
PROCEDURE PrClosePage (pPrPort: TPPrPort);
```

`pPrPort` A pointer to a printing graphics port. (The `TPrPort` record that defines a printing graphics port is described on page 9-51.)

DESCRIPTION

The `PrClosePage` procedure records that you are finished printing the current page. The printer driver can then do whatever it requires (such as releasing temporary memory) to avoid communication difficulties or other problems that may cause the user's computer to crash.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PrClosePage` procedure are

Trap macro	Selector
`_PrGlue`	$1800040C

SEE ALSO

For an example of the use of `PrClosePage`, see Listing 9-2 beginning on page 9-20.

PrPicFile

Use the `PrPicFile` procedure to complete deferred printing.

```
PROCEDURE PrPicFile (hPrint: THPrint; pPrPort: TPPrPort;
                     pIOBuf: Ptr; pDevBuf: Ptr;
                     VAR prStatus: TPrStatus);
```

`hPrint` A handle to a `TPrint` record (described on page 9-44) for a document.

`pPrPort` A pointer to the printing graphics port. (The `TPrPort` record that defines a printing graphics port is described on page 9-51.) If this parameter is `NIL`, the `PrPicFile` procedure allocates a new printing graphics port in a heap.

`pIOBuf` A pointer to an area of memory to be used as an input/output buffer. This parameter should be `NIL` to use the volume buffer for the spool file's volume. If you allocate your own buffer, it must be exactly 522 bytes.

`pDevBuf` A pointer to a device-dependent buffer. This parameter should be `NIL` so that `PrPicFile` allocates a buffer in a heap.

`prStatus` A `TPrStatus` record that `PrPicFile` uses to report on the current page number, current copy, or current file being spooled. You can then display this information to the user. The `TPrStatus` record is described on page 9-49.

DESCRIPTION

The `PrPicFile` procedure sends a file spooled for deferred printing to the printer.

You can determine whether a user has chosen deferred printing by testing for the `bSpoolLoop` constant in the `bJDocLoop` field of the `TPrJob` record contained in the `TPrint` record specified in the `hPrint` parameter. If the `bJDocLoop` field contains the value represented by the `bSpoolLoop` constant, call the `PrPicFile` procedure, which sends the spool file to the printer.

Your application should normally call `PrPicFile` after the `PrCloseDoc` procedure (described on page 9-68).

SPECIAL CONSIDERATIONS

Do not pass, in the `pPrPort` parameter, a pointer to the same printing graphics port you received from the `PrOpenDoc` function (described on page 9-67). If that port was allocated by `PrOpenDoc` itself (that is, if the `pPrPort` parameter to `PrOpenDoc` was `NIL`), then `PrCloseDoc` will already have disposed of the port, making your pointer to it invalid. Of course, if you earlier provided your own storage in `PrOpenDoc`, there's no reason you can't use the same storage again for `PrPicFile`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PrPicFile` procedure are

Trap macro	Selector
`_PrGlue`	$60051480

SEE ALSO

For an example of the use of `PrPicFile`, see Listing 9-2 beginning on page 9-20.

Optimizing Printing

The `PrGeneral` procedure helps you achieve the highest possible resolution on the current printer, verify page orientation, and force enhanced draft-quality printing for printer drivers supporting these options. To select which action you want, pass one of four records to `PrGeneral` and, in a field of that record, you supply the opcode that specifies the action you require. These records are `TGetRslBlk` (described on page 9-53), `TSetRslBlk` (described on page 9-54), `TGetRotnBlk` (described on page 9-56), and `TDftBitsBlk` (described on page 9-55). All of these records are based on the `TGnlData` record (described on page 9-52), so the first three fields of each are identical.

PrGeneral

Use the `PrGeneral` procedure to achieve the highest possible resolution on the current printer, verify page orientation, and allow enhanced draft-quality printing.

```
PROCEDURE PrGeneral (pData: Ptr);
```

`pData` A pointer to one of these four records, depending on your purpose for calling `PrGeneral`:

A `TGetRslBlk` record (described on page 9-53) for determining resolutions of the current printer. You set the `getRslDataOp` opcode in the `iOpCode` field of this record.

A `TSetRslBlk` record (described on page 9-54) for setting the resolution of a `TPrint` record. In the fields of this record, you specify the `setRslOp` opcode, a handle to a `TPrint` record (described on page 9-44), and the new resolutions for the x and y directions.

A `TGetRotnBlk` record (described on page 9-56) when determining whether to print in landscape orientation. You specify the `getRotnOp` opcode and a handle to a `TPrint` record in the fields of this record.

A `TDftBitsBlk` record (described on page 9-55) to use or cancel enhanced draft-quality printing. You specify in the fields of this record either the `draftBitsOp` or `noDraftBitsOp` opcode and a handle to a `TPrint` record.

DESCRIPTION

To select which action you want the `PrGeneral` procedure to undertake, you pass an opcode in the `iOpCode` field of the record that the `pData` parameter points to.

Use the `PrGeneral` procedure with the value `getRslDataOp` in the `iOpCode` field of a `TGetRslBlk` record when you want to determine the resolutions supported by the current printer driver. The `PrGeneral` procedure returns information about the resolutions that the printer driver supports in the `xRslRg`, `yRslRg`, `iRslRecCnt`, and `rgRslRec` fields of the `TGetRslBlk` record.

Use the `PrGeneral` procedure with the value `setRslOp` in the `iOpCode` field of the `TSetRslBlk` record when you want to set the resolution of a `TPrint` record. When called with the `setRslOp` opcode, `PrGeneral` sets the fields relating to x and y resolution in the specified `TPrint` record according to the values of the `iXRsl` and `iYRsl` fields of the `TSetRslBlk` record.

Use the `PrGeneral` procedure with the value `getRotnOp` in the `iOpCode` field of the `TGetRotnBlk` record when you want to determine whether a `TPrint` record specifies landscape orientation. The `PrGeneral` procedure returns in the `fLandscape` field of this record a Boolean value indicating whether the `TPrint` record specifies landscape orientation. When the user chooses landscape orientation from the style dialog box, the `PrStlDialog` function (described on page 9-61) modifies the `TPrint` record accordingly.

Use the `PrGeneral` procedure with the value `draftBitsOp` in the `iOpCode` field of the `TDftBitsBlk` record when you want to use enhanced draft-quality printing. Typically, you use enhanced draft-quality printing when you want to print bitmaps as well as text in a draft-quality printout on an ImageWriter printer. Use the `noDraftBitsOp` opcode to cancel the use of enhanced draft-quality printing.

If you want to force enhanced draft-quality printing, you should call `PrGeneral` with the `draftBitsOp` opcode before displaying the print dialog boxes to the user. Use of the `draftBitsOp` opcode may cause the printer driver to make some items in its print dialog boxes inactive; for example, the ImageWriter printer driver makes the landscape icon in the style dialog box (landscape printing is not available for draft-quality printing) and the Best and Faster buttons in the job dialog box inactive.

The `PrGeneral` procedure returns error information in the `iError` field of each of these records. You should check the value in the `iError` field after each use of `PrGeneral`. You should also use the `PrError` function (which returns the result code left by the last Printing Manager routine) after checking the `iError` field, to be sure that no additional errors were generated. If `PrError` returns the result code `resNotFound` after you call `PrGeneral`, then the current printer driver doesn't support `PrGeneral`. You should clear the error by calling the `PrSetError` procedure and passing `noErr` in its parameter; otherwise, `PrError` might still contain this error the next time you check it. (The `PrError` function and the `PrSetError` procedure are described on page 9-75 and page 9-78, respectively.)

SPECIAL CONSIDERATIONS

If you call `PrGeneral` with the `draftBitsOp` opcode after using the `PrJobDialog` or `PrDlgMain` function, and if the user chooses draft printing from the job dialog box, the ImageWriter does not print any bitmaps or pixel maps contained in the document.

Enhanced draft-quality printing is of limited usefulness, as described in "Enhancing Draft-Quality Printing" on page 9-33.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PrGeneral` procedure are

Trap macro	Selector
`_PrGlue`	$70070480

RESULT CODES

`opNotImpl`	2	Printer driver does not support this opcode
`noSuchRsl`	1	Requested resolution not supported by the currently selected printer
`noErr`	0	No error

SEE ALSO

See Listing 9-4 on page 9-31 for an example of how to use the `getRslDataOp` opcode to determine what printer resolutions are available for the current printer. The same listing shows an example of how to use the `setRslOp` opcode to set the resolution for the current printer.

See Listing 9-5 on page 9-33 for an example of using the `getRotnOp` opcode to determine if the user has selected landscape orientation.

See "Enhancing Draft-Quality Printing" on page 9-33 for more information on using the `draftBitsOp` and `noDraftBitsOp` opcodes to force the use of or to cancel the use of enhanced draft-quality printing.

Handling Printing Errors

The `PrError` function returns the result code reported by the last Printing Manager routine. The `PrSetError` procedure lets you set the value of the current Printing Manager error.

PrError

You can use the `PrError` function to get the result code returned by the last Printing Manager routine.

```
FUNCTION PrError: Integer;
```

DESCRIPTION

The `PrError` function returns the error reported by the last Printing Manager routine. If an error that does not belong to the Printing Manager occurs during the printing process, the Printing Manager puts it into low memory, where it can be retrieved with a call to `PrError`. The Printing Manager then terminates the printing loop if necessary. If you encounter an error in the middle of a printing loop, do not end printing abruptly; call the close routines for any open routines you have already made and let the Printing Manager terminate properly.

Do not display any alert or dialog boxes to report an error until the end of the printing loop. Once at the end, check for the error again; if there is no error, assume that printing completed normally. If the error is still present, then you can alert the user.

The most common error encountered is `PAPNoPrinter`, which is usually generated if no printer is selected. Since this error is so common, it is a good idea to create and display an alert box asking the user to select a printer from the Chooser when this error is encountered.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PrError` function are

Trap macro	Selector
`_PrGlue`	$BA000000

RESULT CODES

iPrAbort	128	Application or user requested cancel
opNotImpl	2	Requested PrGeneral opcode not implemented in the current printer driver
noSuchRsl	1	Resolution requested with the PrGeneral procedure is not supported
noErr	0	No error
iPrSavPFil	–1	Problem saving print file
controlErr	–17	Unimplemented control instructions; the Device Manager returns this result code
iIOAbort	–27	I/O error
iMemFullErr	–108	There is not enough room in the heap zone
resNotFound	–192	The current printer driver does not support PrGeneral (described on page 9-72); you should clear this error with a call to PrSetError (described in the next section) with a parameter value of 0; otherwise, PrError might still contain this error the next time you check it

The following result codes are specific to the LaserWriter 8 printer driver:

PAPNoCCBs	–4096	There are no free connect control blocks (CCBs) available
PAPBadRefnum	–4097	Bad connection reference number
PAPActive	–4098	The request is already active
PAPTooBig	–4099	The write request is too big
PAPConnClosed	–4100	The connection is closed
PAPNoPrinter	–4101	The printer is not found, is closed, or is not selected
	–8131	Printer not responding
manualFeedTOErr	–8132	A timeout occurred (that is, no communication has occurred with the printer for two minutes); this is usually caused by an extremely long imaging time or a dropped connection
generalPSErr	–8133	A PostScript error occurred during transmission of data to the printer; this is most often caused by a bug in the application-supplied PostScript code
zoomRangeErr	–8160	The print image enlarged by the user with the Page Setup dialog box overflows the available page resolution
errBadFontKeyType	–8976	Font found in printer is not Type 1, TrueType, or bitmapped font
errPSStateUnderflow	–8977	PostScript stack underflow while restoring graphics state
errNoPattern	–8978	The pixel pattern could not be found and could not be built
errBadConverterID	–8979	The 'PDEF' converter doesn't exist

`errNoPagesSpooled`	–8980	Application called `PrOpenDoc` and `PrCloseDoc` without calling `PrOpenPage` and `PrClosePage` in between
`errNullColorInfo`	–8981	The `getColor` function called with null `GetColorInfo` handle
`errPSFileNameNull`	–8982	The filename pointer for the spool file is null
`errSpoolFolderIsAFile`	–8983	The spool folder is a file instead of a folder
`errBadConverterIndex`	–8984	When saving a spool file to disk, the value `fileTypeIndex` had no matching entry in the driver
`errDidNotDownloadFont`	–8985	A PostScript outline could not be found for a PostScript font, and there is no associated `'sfnt'` resource
`errBitmapFontMissing`	–8986	Unable to build bitmap for font
`errPSFileName`	–8987	PostScript file isn't named
`errCouldNotMakeNumberedFilename`	–8989	Could not make a unique filename for the spool file
`errBadSpoolFileVersion`	–8990	Bad version number in header of spool file
`errNoProcSetRes`	–8991	The resource describing needed procedure sets is unavailable for the PostScript prolog
`errInLineTimeout`	–8993	The printer is not responding
`errUnknownPSLevel`	–8994	The PostScript level has an unknown value
`errFontNotFound`	–8995	Font query reply didn't match any fonts in list of PostScript names
`errSizeListBad`	–8996	The size list contained an entry that could not be reconciled with the typeface list
`errFaceListBad`	–8997	Entry could not be found in typeface list
`errNotAKey`	–8998	Key for desired font number and style could not be found in font table

SEE ALSO

See "Handling Printing Errors" on page 9-41 for more information on using `PrError`. See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information about displaying alert and dialog boxes.

PrSetError

You can use the `PrSetError` procedure to set the value of the current printing error.

```
PROCEDURE PrSetError (iErr: Integer);
```

`iErr` The result to set as the current printing error.

DESCRIPTION

The `PrSetError` procedure stores the specified value into the global variable `PrintErr`, where the Printing Manager keeps its result code. You can use `PrSetError` to cancel a printing operation.

ASSEMBLY-LANGUAGE INFORMATION

You should not directly access the location of the global variable `PrintErr`; instead you should use the `PrError` function or `PrSetError` procedure to get the value of this variable.

The trap macro and routine selector for the `PrSetError` procedure are

Trap macro	Selector
`_PrGlue`	$C0000200

Low-Level Routines

Low-level routines are available for use when printing on some ImageWriter printers. (The ImageWriter LQ driver does not support these routines.) However, Apple strongly discourages you from using these routines—with the exception of the `PrDrvrVers` function. The others are documented here only for completeness.

Instead of using the low-level routines, you should use the high-level routines of the Printing Manager. Low-level routines are not guaranteed to work in precisely the same manner in future versions of the system software. Low-level routines are primarily suited for functions such as text streaming (the process of receiving data from a source and printing it immediately, without any intermediate formatting). In addition, if you use the low-level routines and the user prints a document on a LaserWriter printer, the LaserWriter printer driver translates all calls to low-level routines to the matching high-level routines, so your application does not gain a speed advantage.

▲ **WARNING**
Apple strongly discourages you from using these routines. If you do, do not mix high-level routines and low-level routines after opening the printer driver. The only exception to this is that you may use the `PrDrvrVers` function (described next) with the high-level routines. ▲

PrDrvrVers

You can use the `PrDrvrVers` function to determine the version of the printer driver for the current printer.

```
FUNCTION PrDrvrVers: Integer;
```

DESCRIPTION

The `PrDrvrVers` function returns the version number of the printer driver for the current printer. This is the only low-level printing function you may call from the high-level interface.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PrDrvrVers` function are

Trap macro	Selector
`_PrGlue`	$9A000000

PrDrvrOpen

You can use the `PrDrvrOpen` procedure to open the current printer driver.

```
PROCEDURE PrDrvrOpen;
```

DESCRIPTION

The `PrDrvrOpen` procedure opens the printer driver, reading it into memory if necessary.

SPECIAL CONSIDERATIONS

Use the `PrDrvrOpen` procedure with the `PrDrvrClose` procedure (described in the next section). Do not mix these procedures with the `PrOpen` and `PrClose` procedures (described on page 9-57 and page 9-58, respectively).

Apple strongly discourages you from using this routine.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PrDrvrOpen` procedure are

Trap macro	Selector
`_PrGlue`	$80000000

PrDrvrClose

You can use the `PrDrvrClose` procedure to close the printer driver.

```
PROCEDURE PrDrvrClose;
```

DESCRIPTION

The `PrDrvrClose` procedure closes the printer driver, releasing the memory it occupies.

SPECIAL CONSIDERATIONS

Use the `PrDrvrClose` procedure with the `PrDrvrOpen` procedure (described in the previous section). Do not mix these procedures with the `PrOpen` and `PrClose` procedures (described on page 9-57 and page 9-58, respectively).

Apple strongly discourages you from using this routine.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PrDrvrClose` procedure are

Trap macro	Selector
`_PrGlue`	$88000000

PrDrvrDCE

You can use the `PrDrvrDCE` function to get a handle to the current printer driver's device control entry (DCE).

```
FUNCTION PrDrvrDCE: Handle;
```

DESCRIPTION

The `PrDrvrDCE` function returns a handle to the current printer driver's DCE. A printer driver's DCE contains specific information about that printer driver. You can also get a handle to the driver's DCE by calling the Device Manager function `GetDCtlEntry`.

SPECIAL CONSIDERATIONS

Apple strongly discourages you from using this routine.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PrDrvrDCE` function are

Trap macro	Selector
`_PrGlue`	$94000000

SEE ALSO

For more information about DCEs and how the Device Manager uses them, see *Inside Macintosh: Devices*.

PrCtlCall

You can use the `PrCtlCall` procedure to send various requests to the current printer driver.

```
PROCEDURE PrCtlCall (iWhichCtl: Integer; lParam1: LongInt;
                     lParam2: LongInt; lParam3: LongInt);
```

`iWhichCtl` A value that indicates the operation to perform. You can use these constants in this parameter:

```
CONST
  iPrBitsCtl     = 4;  {print a bitmap object}
  iPrIOCtl       = 5;  {perform text streaming}
  iPrEvtCtl      = 6;  {print object specified in }
                       { lParam1 parameter}
  iPrDevCtl      = 7;  {device control command}
```

`lParam1` The use of this parameter varies according to the value in the `iWhichCtl` parameter. See the following paragraphs for this information.

`lParam2` The use of this parameter varies according to the value in the `iWhichCtl` parameter. See the following paragraphs for this information.

`lParam3` The use of this parameter varies according to the value in the `iWhichCtl` parameter. See the following paragraphs for this information.

DESCRIPTION

The `PrCtlCall` procedure performs the operation indicated by the `iWhichCtl` parameter. Depending on the operation, `PrCtlCall` may also use information in the `lParam1`, `lParam2`, and `lParam3` parameters. The `PrCtlCall` procedure calls the printer driver's control routine. Instead of sending the low-level calls to the printer driver, the `PrCtlCall` procedure converts the call into its high-level equivalent before execution.

You can use the `PrCtlCall` procedure with the `iPrBitsCtl` control constant when you want to print bitmaps. In this case, you should supply the parameters to `PrCtlCall` with the following information:

`iWhichCtl` The constant `iPrBitsCtl`. This constant allows you to send all or part of a QuickDraw bitmap directly to the printer.

`lParam1` A pointer to the QuickDraw bitmap to print.

`lParam2` A pointer to the rectangle you want to print.

`lParam3` The type of resolution used to print the bitmap. The LaserWriter printer driver ignores this flag. This parameter can have one of the following values:

Constant	Value	Description
`lScreenBits`	$00000000	The resolution is 80 by 72 dpi
`lPaintBits`	$00000001	The resolution is 72 by 72 dpi
`lHiScreenBits`	$00000002	The resolution is 160 by 144 dpi
`lHiPaintBits`	$00000003	The resolution is 144 by 144 dpi

You can use the `PrCtlCall` procedure with the `iPrIOCtl` control constant when you want text streaming in your application. (Text streaming is useful for fast printing of text when speed is more important than visual fidelity or formatting. It makes no use of QuickDraw.) In this case, you should supply the parameters to `PrCtlCall` with the following information:

`iWhichCtl` The constant `iPrIOCtl`. This constant causes text streaming to occur.

`lParam1` A pointer to the beginning of the text.

`lParam2` The number of bytes to transfer. The high-order word must be 0.

`lParam3` This should be 0.

You can use the `PrCtlCall` procedure with the `iPrEvtCtl` control constant for printing the screen or the frontmost window on most ImageWriter printers. (The LaserWriter printer driver does not support this call.) In this case, you should supply the parameters to `PrCtlCall` with the following information:

`iWhichCtl` The constant `iPrEvtCtl`. This constant prints the object you have selected using the `lParam1` parameter.

`lParam1` This parameter selects the object to be printed. If this value is $00000000, you want to print the screen. If this value is $00010000, you want to print the frontmost window.

lParam2 This should be NIL.

lParam3 This should be NIL.

You can use the PrCtlCall procedure with the iPrDevCtl control constant for controlling the printer device. In this case, you should supply the parameters to PrCtlCall with the following information:

iWhichCtl The constant iPrDevCtl.

lParam1 The action you want to take. The values possible for this parameter are listed in Table 9-1.

lParam2 This should be NIL.

lParam3 This should be NIL.

Table 9-1 Values for the lParam1 parameter when using the iPrDevCtl control constant

Constant	Value	Description
lPrDocOpen	$00010000	Opens the document. This is similar to the high-level routine PrOpenDoc and should be followed with a call to PrCtlCall with the iPrDetl control call and an lParam1 value of lPrDocClose.
lPrReset	$00010000	Reserved.
lPrPageClose	$00020000	Closes the page. This is similar to the high-level routine PrClosePage and should follow a call to PrCtlCall with the iPrDevCtl control call and an lParam1 value of lPrPageOpen.
lPrPageEnd	$00020000	Same as lPrPageClose.
lPrLineFeed	$00030000	Paper advance.
lPrLFStd	$0003FFFF	Carriage return with line feed. The ImageWriter printer driver causes a carriage return plus a paper feed of one-sixth of an inch. The LaserWriter printer driver moves the pen location down the page.
lPrPageOpen	$00040000	Opens the page for printing. This is similar to the high-level routine PrOpenPage and should be followed with a call to PrCtlCall with the iPrDevCtl control call and an lParam1 value of lPrPageClose.
lPrDocClose	$00050000	Closes the document. This is similar to the high-level routine PrCloseDoc and should follow a call to PrCtlCall with the iPrDevCtl control call and an lParam1 value of lPrDocOpen.

SPECIAL CONSIDERATIONS

Apple strongly discourages you from using this routine.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PrCtlCall` procedure are

Trap macro	Selector
`_PrGlue`	$A0000E00

Application-Defined Routines

The printer driver for the current printer periodically calls an idle procedure while sending a document to the printer. You can provide your own idle procedure (here called `MyDoPrintIdle`) that handles events in a dialog box reporting the status of the print job.

If you add items—such as checkboxes and radio buttons—to the default style or job dialog box, your application uses the `PrDlgMain` function to display the dialog box. In one of the parameters to `PrDlgMain`, you pass the address of an initialization function (here called `MyPrDialogAppend`) that you use to append items to your dialog box.

If you append items to the style or job dialog boxes, you need to provide a function (sometimes called a *dialog hook*) to handle events in these items. You should also provide an event filter function to handle events that the Dialog Manager doesn't handle—such as update events in background windows—in modal dialog boxes. For a description of how to handle events in dialog boxes and how to write an event filter function for modal dialog boxes, see the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

MyDoPrintIdle

The printer driver for the current printer periodically calls an idle procedure while sending a document to the printer. The Printing Manager's default idle procedure allows the user to cancel printing. This procedure polls the keyboard and sets the `iPrAbort` result code if the user presses Command-period to cancel the print job. However, the default idle procedure does not display a print status dialog box. It is up to the printer driver or your application to display a print status dialog box.

Most printer drivers display their own status dialog boxes. However, your application can display its own status dialog box that reports the current status of the printing operation to the user. If it does, your status dialog box should allow the user to press Command-period to cancel the printing operation, and it may also provide a button allowing the user to cancel the printing operation. To handle update events in your status dialog box, Command-period keyboard events, and clicks in your Cancel button (if you provide one), you should provide your own idle procedure.

```
PROCEDURE MyDoPrintIdle;
```

DESCRIPTION

As described in "Writing an Idle Procedure" beginning on page 9-38, you install your idle procedure in the `pIdleProc` field of the `TPrint` record. The printer driver runs your idle procedure periodically. It stops running once the entire document has been sent to the printer and does not run while the printer actually prints. The idle procedure takes no parameters and returns no result.

SEE ALSO

See Figure 9-9 on page 9-14 for an example of an application-defined status dialog box. Listing 9-9 on page 9-40 illustrates an idle procedure. See "Writing an Idle Procedure" beginning on page 9-38 for complete information about providing your own idle procedure.

MyPrDialogAppend

If you customize a style or job dialog box, your application uses the `PrDlgMain` function to display the dialog box. In one of the parameters to `PrDlgMain`, you pass the address of an initialization function that you use to append items—such as checkboxes and radio buttons—to the dialog box. Here is how might declare your initialization function if you were to name it `MyPrDialogAppend`:

```
FUNCTION MyPrDialogAppend (hPrint: THPrint): TPPrDlg;
```

`hPrint` A handle to a `TPrint` record (described on page 9-44).

DESCRIPTION

Your `MyPrDialogAppend` function should use the Dialog Manager procedure `AppendDITL` to add items to the style or job dialog box for the document whose `TPrint` record is passed in the `hPrint` parameter. As its function result, your function should return a pointer to the `TPrDlg` record (described on page 9-50) for the customized style or job dialog box.

You can use the `PrStlInit` or `PrJobInit` function (described on page 9-64 and page 9-65, respectively) to get an initialized `TPrDlg` record for the current printer.

Your `MyPrDialogAppend` function should install pointers to two functions in the `TPrDlg` record for this dialog box. Put a pointer to one function in the `pFltrProc` field; this function should handle events (such as update events in background applications and disk-inserted events) that the Dialog Manager doesn't handle in a modal dialog box. Put a pointer to the second function in the `pItemProc` field; this function should handle events, such as mouse clicks, in the items added to the dialog box.

SEE ALSO

Listing 9-8 on page 9-37 shows an example of the `MyPrDialogAppend` function; Listing 9-7 on page 9-37 shows how to pass the address of this function to the `PrDlgMain` function. See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information about the `AppendDITL` procedure and about handling events in dialog boxes.

Summary of the Printing Manager

Pascal Summary

Constants

```
CONST
   iPrPgFst       = 1;        {page range constant--first page}
   iPrRelease     = 3;        {current version number of the printer driver}
   iPrPgFract     = 120;      {page scale factor}
   iPFMaxPgs      = 128;      {maximum pages in spool file}
   iPrPgMax       = 9999;     {page range constant--last page}

   {PrCtlCall constants for the iWhichCtl parameter}
   iPrBitsCtl     = 4;        {print a bitmap object}
   iPrIOCtl       = 5;        {perform text streaming}
   iPrEvtCtl      = 6;        {print object specified in lParam1 parameter}
   iPrDevCtl      = 7;        {device control command}

   {constants used with iPrBitsCtl (in the lParam1 parameter of PrCtlCall)}
   lScreenBits    = $00000000;   {resolution is 80 x 72 dpi}
   lPaintBits     = $00000001;   {resolution is 72 x 72 dpi}
   lHiScreenBits  = $00000002;   {resolution is 160 x 144 dpi}
   lHiPaintBits   = $00000003;   {resolution is 144 x 144 dpi}

   {constants used with iPrEvtCtl (in the lParam3 parameter of PrCtlCall)}
   lPrEvtAll      = $0002FFFD;   {the entire screen}
   lPrEvtTop      = $0001FFFD;   {the frontmost window}

   {constants used with iPrDevCtl (in the lParam1 parameter of PrCtlCall)}
   lPrReset       = $00010000;   {reserved}
   lPrLineFeed    = $00030000;   {paper advance}
   lPrLFStd       = $0003FFFF;   {carriage return with line feed}
   lPrLFSixth     = $0003FFFF;   {used for low-level call for ImageWriter}
   lPrPageEnd     = $00020000;   {end page}
   lPrDocOpen     = $00010000;   {open document for printing}
   lPrPageOpen    = $00040000;   {open page for printing}
   lPrPageClose   = $00020000;   {close page for printing}
```

```
   lPrDocClose    = $00050000;   {close document for printing}

   bDraftLoop     = 0;        {draft-quality printing}
   bSpoolLoop     = 1;        {deferred printing}
   bUser1Loop     = 2;        {reserved}
   bUser2Loop     = 3;        {reserved}

   iPrSavPFil     = -1;       {problem saving print file}
   iPrAbort       = $0080;    {the user pressed Command-period}

   iFMgrCtl    = 8;           {File Mgr's dialog-hook proc's control number}
   pPrGlobals  = $00000944;   {PrVars low memory area}

   iPrDrvrRef     = -3;       {reference number of printer driver}

   {opcodes used with PrGeneral}
   getRslDataOp   = 4;        {get resolutions for the current printer}
   setRslOp       = 5;        {set resolutions for a TPrint record}
   draftBitsOp    = 6;        {force enhanced draft-quality printing}
   noDraftBitsOp  = 7;        {cancel enhanced draft-quality printing}
   getRotnOp      = 8;        {get page orientation of a TPrint record}

   {result codes from PrGeneral}
   noSuchRsl      = 1;        {resolution not supported}
```

Data Types

```
TYPE
   TPPrint = ^TPrint;         {pointer to a TPrint record}
   THPrint = ^TPPrint;        {handle to a TPrint record}
   TPrint =                   {print record}
   RECORD
      iPrVersion: Integer;    {reserved}
      prInfo:     TPrInfo;    {resolution of device & page rectangle}
      rPaper:     Rect;       {paper rectangle}
      prStl:      TPrStl;     {printer driver number & feed type}
      prInfoPT:   TPrInfo;    {reserved}
      prXInfo:    TPrXInfo;   {reserved}
      prJob:      TPrJob;     {information from the job dialog box}
      printX:     ARRAY[1..19] OF Integer;
                              {reserved}
   END;
```

```
TPPrInfo = ^TPrInfo;
TPrInfo =                   {printer information record}
RECORD
   iDev:    Integer;        {reserved}
   iVRes:   Integer;        {vertical resolution of printer, in dpi}
   iHRes:   Integer;        {horizontal resolution of printer, in dpi}
   rPage:   Rect;           {the page rectangle}
END;

TPPrJob = ^TPrJob;
TPrJob =                       {print job record}
RECORD
   iFstPage:   Integer;        {first page of page range}
   iLstPage:   Integer;        {last page of page range}
   iCopies:    Integer;        {number of copies}
   bJDocLoop:  SignedByte;     {printing method: draft or deferred}
   fFromUsr:   Boolean;        {reserved}
   pIdleProc:  PrIdleProcPtr; {pointer to an idle procedure}
   pFileName:  StringPtr;      {spool filename: NIL for default}
   iFileVol:   Integer;        {spool file volume; set to 0 initially}
   bFileVers:  SignedByte;     {spool file version; set to 0 initially}
   bJobX:      SignedByte;     {reserved}
END;

TPPrStl = ^TPrStl;
TPrStl =                    {printing style record}
RECORD
   wDev:    Integer;        {device number of printer}
   iPageV:  Integer;        {reserved}
   iPageH:  Integer;        {reserved}
   bPort:   SignedByte;     {reserved}
   feed:    TFeed;          {feed type}
END;

TPPrStatus = ^TPrStatus;
TPrStatus =                 {printing status record}
RECORD
   iTotPages:  Integer;     {total pages in print file}
   iCurPage:   Integer;     {current page number}
   iTotCopies: Integer;     {total copies requested}
   iCurCopy:   Integer;     {current copy number}
   iTotBands:  Integer;     {reserved}
   iCurBand:   Integer;     {reserved}
   fPgDirty:   Boolean;     {TRUE if current page has been written to}
```

```
   fImaging:   Boolean;    {reserved}
   hPrint:     THPrint;    {handle to the active TPrint record}
   pPrPort:    TPPrPort;   {pointer to the active printing graphics port}
   hPic:       PicHandle;  {handle to the active picture}
END;

TPPrDlg = ^TPrDlg;
TPrDlg =                            {print dialog box record}
RECORD
   Dlg:        DialogRecord;        {a dialog record}
   pFltrProc:  ModalFilterProcPtr;  {pointer to event filter}
   pItemProc:  PItemProcPtr;        {pointer to item-handling function}
   hPrintUsr:  THPrint;             {handle to a TPrint record}
   fDoIt:      Boolean;             {TRUE means user clicked OK}
   fDone:      Boolean;             {TRUE means user clicked OK or Cancel}
   lUser1:     LongInt;             {storage for your application}
   lUser2:     LongInt;             {storage for your application}
   lUser3:     LongInt;             {storage for your application}
   lUser4:     LongInt;             {storage for your application}
END;

TPPrPort = ^TPrPort;
TPrPort =                  {printing graphics port record}
RECORD
   gPort:      GrafPort;   {graphics port for printing}
   gProcs:     QDProcs;    {procedures for printing in the graphics port}
   lGParam1:   LongInt;    {reserved}
   lGParam2:   LongInt;    {reserved}
   lGParam3:   LongInt;    {reserved}
   lGParam4:   LongInt;    {reserved}
   fOurPtr:    Boolean;    {reserved}
   fOurBits:   Boolean;    {reserved}
END;

TFeed = (feedCut,feedFanfold,feedMechCut,feedOther);

TScan = (scanTB,scanBT,scanLR,scanRL);

TPRect = ^Rect;

PrIdleProcPtr = ProcPtr;

PItemProcPtr = ProcPtr;
```

```
   PDlgInitProcPtr = ProcPtr;

{records used by PrGeneral}

   TGnlData =
   RECORD
      iOpCode:    Integer; {opcode passed to PrGeneral}
      iError:     Integer; {result code returned by PrGeneral}
      lReserved:  LongInt; {more fields here depending on opcode}
   END;

   TGetRslBlk =               {get-resolution record}
   RECORD
      iOpCode:    Integer;    {the getRslDataOp opcode}
      iError:     Integer;    {result code returned by PrGeneral}
      lReserved:  LongInt;    {reserved}
      iRgType:    Integer;    {printer driver version number}
      xRslRg:     TRslRg;     {x-direction resolution range}
      yRslRg:     TRslRg;     {y-direction resolution range}
      iRslRecCnt: Integer;    {number of resolution records}
      rgRslRec:   ARRAY[1..27] OF TRslRec;
                              {array of resolution records}
   END;

   TRslRg =
   RECORD
      iMin:    Integer; {minimum resolution supported}
      iMax:    Integer; {maximum resolution supported}
   END;

   TRslRec =
   RECORD
      iXRsl:   Integer; {discrete resolution, x direction}
      iYRsl:   Integer; {discrete resolution, y direction}
   END;

   TSetRslBlk =               {set-resolution record}
   RECORD
      iOpCode:    Integer;    {the setRslOp opcode}
      iError:     Integer;    {result code returned by PrGeneral}
      lReserved:  LongInt;    {reserved}
      hPrint:     THPrint;    {handle to the current TPrint record}
      iXRsl:      Integer;    {x-direction resolution you want}
      iYRsl:      Integer;    {y-direction resolution you want}
   END;
```

```
TDftBitsBlk =              {draft bits record}
RECORD
   iOpCode:    Integer;    {draftBitsOp or noDraftBitsOp opcode}
   iError:     Integer;    {result code returned by PrGeneral}
   lReserved:  LongInt;    {reserved}
   hPrint:     THPrint;    {handle to the current TPrint record}
END;

TGetRotnBlk =              {page orientation record}
RECORD
   iOpCode:    Integer;    {the getRotnOp opcode}
   iError:     Integer;    {result code returned by PrGeneral}
   lReserved:  LongInt;    {reserved}
   hPrint:     THPrint;    {handle to current TPrint record}
   fLandscape: Boolean;    {TRUE if user selected landscape printing}
   bXtra:      SignedByte; {reserved}
END;
```

Printing Manager Routines

Opening and Closing the Printing Manager

```
PROCEDURE PrOpen;
PROCEDURE PrClose;
```

Initializing and Validating TPrint Records

```
PROCEDURE PrintDefault      (hPrint: THPrint);
FUNCTION PrValidate         (hPrint: THPrint): Boolean;
```

Displaying and Customizing the Print Dialog Boxes

```
FUNCTION PrStlDialog        (hPrint: THPrint): Boolean;
FUNCTION PrJobDialog        (hPrint: THPrint): Boolean;
FUNCTION PrDlgMain          (hPrint: THPrint; pDlgInit: PDlgInitProcPtr):
                             Boolean;
FUNCTION PrStlInit          (hPrint: THPrint): TPPrDlg;
FUNCTION PrJobInit          (hPrint: THPrint): TPPrDlg;
PROCEDURE PrJobMerge        (hPrintSrc: THPrint; hPrintDst: THPrint);
```

Printing a Document

```
FUNCTION PrOpenDoc          (hPrint: THPrint; pPrPort: TPPrPort;
                             pIOBuf: Ptr): TPPrPort;
PROCEDURE PrCloseDoc        (pPrPort: TPPrPort);
PROCEDURE PrOpenPage        (pPrPort: TPPrPort; pPageFrame: TPRect);
PROCEDURE PrClosePage       (pPrPort: TPPrPort);
PROCEDURE PrPicFile         (hPrint: THPrint; pPrPort: TPPrPort;
                             pIOBuf: Ptr; pDevBuf: Ptr;
                             VAR prStatus: TPrStatus);
```

Optimizing Printing

```
PROCEDURE PrGeneral         (pData: Ptr);
```

Handling Printing Errors

```
FUNCTION PrError            : Integer;
PROCEDURE PrSetError        (iErr: Integer);
```

Low-Level Routines

```
FUNCTION PrDrvrVers         : Integer;
PROCEDURE PrDrvrOpen;
PROCEDURE PrDrvrClose;
FUNCTION PrDrvrDCE          : Handle;
PROCEDURE PrCtlCall         (iWhichCtl: Integer; lParam1: LongInt;
                             lParam2: LongInt; lParam3: LongInt);
```

Application-Defined Routines

```
PROCEDURE MyDoPrintIdle;
FUNCTION MyPrDialogAppend   (hPrint: THPrint): TPPrDlg;
```

C Summary

Constants

```
enum {
   iPrPgFst    = 1,     /* page range constant--first page */
   iPrRelease  = 3,     /* current version number of the printer driver */
   iPrPgFract  = 120,   /* page scale factor */
   iPFMaxPgs   = 128,   /* maximum pages in spool file */
   iPrPgMax    = 9999,  /* page range constant--last page */

   /* PrCtlCall constants for the iWhichCtl parameter */
   iPrBitsCtl     = 4,  /* print a bitmap object */
   iPrIOCtl       = 5,  /* perform text streaming */
   iPrEvtCtl      = 6,  /* print object specified in lParam1 parameter */
   iPrDevCtl      = 7,  /* device control command */

   /* constants used with iPrBitsCtl (in the lParam1 parameter
      of PrCtlCall) */
   lScreenBits    = 0,           /* resolution is 80 x 72 dpi */
   lPaintBits     = 1,           /* resolution is 72 x 72 dpi */
   lHiScreenBits  = 0x00000002,  /* resolution is 160 x 144 dpi */
   lHiPaintBits   = 0x00000003,  /* resolution is 144 x 144 dpi */

   /* constants used with iPrEvtCtl (in the lParam3 parameter
      of PrCtlCall) */
   lPrEvtAll      = 0x0002FFFD,  /* the entire screen */
   lPrEvtTop      = 0x0001FFFD,  /* the frontmost window */

   /* constants used with iPrDevCtl (in the lParam1 parameter
      of PrCtlCall) */
   lPrReset       = 0x00010000,  /* reserved */
   lPrLineFeed    = 0x00030000,  /* paper advance */
   lPrLFStd       = 0x0003FFFF,  /* carriage return with line feed */
   lPrLFSixth     = 0x0003FFFF,  /* used for low-level call for
                                    ImageWriter */
   lPrPageEnd     = 0x00020000,  /* end page */
   lPrDocOpen     = 0x00010000,  /* open document for printing */
   lPrPageOpen    = 0x00040000,  /* open page for printing */
   lPrPageClose   = 0x00020000,  /* close page for printing */
   lPrDocClose    = 0x00050000,  /* close document for printing */
```

```
   bDraftLoop     = 0,          /* draft-quality printing */
   bSpoolLoop     = 1,          /* deferred printing */
   bUser1Loop     = 2,          /* reserved */
   bUser2Loop     = 3,          /* reserved */

   iPrSavPFil     = -1,         /* problem saving print file */
   iPrAbort       = 0x0080,     /* the user pressed Command-period */

   iFMgrCtl       = 8,          /* File Mgr's dialog-hook proc's control
                                   number */
   pPrGlobals     = 0x00000944,     /* PrVars low memory area */

   iPrDrvrRef     = -3, /* reference number of printer driver */

   /* opcodes used with PrGeneral */
   getRslDataOp   = 4,  /* get resolutions for the current printer */
   setRslOp       = 5,  /* set resolutions for a TPrint record */
   draftBitsOp    = 6,  /* force enhanced draft-quality printing */
   noDraftBitsOp  = 7,  /* cancel enhanced draft-quality printing */
   getRotnOp      = 8,  /* get page orientation of a TPrint record */

   /* result code from PrGeneral */
   noSuchRsl      = 1,  /* resolution not supported */
};
```

Data Types

```
struct TPrint {         /* print record */
   short    iPrVersion; /* reserved */
   TPrInfo  prInfo;     /* resolution of device & page rectangle */
   Rect     rPaper;     /* paper rectangle */
   TPrStl   prStl;      /* printer driver number & feed type */
   TPrInfo  prInfoPT;   /* reserved */
   TPrXInfo prXInfo;    /* reserved */
   TPrJob   prJob;      /* information from the job dialog box */
   short    printX[19]; /* reserved */
};
typedef struct TPrint TPrint;
typedef TPrint *TPPrint, **THPrint;

struct TPrInfo {        /* printer information record */
   short    iDev;       /* reserved */
   short    iVRes;      /* vertical resolution of printer, in dpi */
```

```
   short    iHRes;      /* horizontal resolution of printer, in dpi */
   Rect     rPage;      /* the page rectangle */
};
typedef struct TPrInfo TPrInfo;
typedef TPrInfo *TPPrInfo;

struct TPrJob {         /* print job record */
   short    iFstPage;   /* first page of page range */
   short    iLstPage;   /* last page of page range */
   short    iCopies;    /* number of copies */
   char     bJDocLoop;  /* printing method: draft or deferred */
   Boolean  fFromUsr;   /* reserved */
   PrIdleProcPtr
            pIdleProc;  /* pointer to an idle procedure */
   StringPtr
            pFileName;  /* spool filename: NIL for default */
   short    iFileVol;   /* spool file volume; set to 0 initially */
   char     bFileVers;  /* spool file version; set to 0 initially */
   char     bJobX;      /* reserved */
};
typedef struct TPrJob TPrJob;
typedef TPrJob *TPPrJob;

struct TPrStl {      /* printing style record */
   short    wDev;    /* device number of printer */
   short    iPageV;  /* reserved */
   short    iPageH;  /* reserved */
   char     bPort;   /* reserved */
   TFeed    feed;    /* feed type */
};
typedef struct TPrStl TPrStl;
typedef TPrStl *TPPrStl;

struct TPrStatus {         /* printing status record */
   short       iTotPages;  /* total pages in print file */
   short       iCurPage;   /* current page number */
   short       iTotCopies; /* total copies requested */
   short       iCurCopy;   /* current copy number */
   short       iTotBands;  /* reserved */
   short       iCurBand;   /* reserved */
   Boolean     fPgDirty;   /* TRUE if current page has been written to */
   Boolean     fImaging;   /* reserved */
   THPrint     hPrint;     /* handle to the active TPrint record */
```

```
   TPPrPort    pPrPort;    /* pointer to the active printing graphics port */
   PicHandle   hPic;       /* handle to the active picture */
};
typedef struct TPrStatus TPrStatus;
typedef TPrStatus *TPPrStatus;

struct TPrDlg {               /* print dialog box record */
   DialogRecord   Dlg;        /* a dialog record */
   ModalFilterProcPtr
                  pFltrProc;  /* pointer to event filter */
   PItemProcPtr   pItemProc;  /* pointer to item-handling function */
   THPrint        hPrintUsr;  /* handle to a TPrint record */
   Boolean        fDoIt;      /* TRUE means user clicked OK */
   Boolean        fDone;      /* TRUE means user clicked OK or Cancel */
   long           lUser1;     /* storage for your application */
   long           lUser2;     /* storage for your application */
   long           lUser3;     /* storage for your application */
   long           lUser4;     /* storage for your application */
};
typedef struct TPrDlg TPrDlg;
typedef TPrDlg *TPPrDlg;

typedef pascal TPPrDlg (*PDlgInitProcPtr)(THPrint hPrint);

truct TPrPort {         /* printing graphics port record */
   GrafPort gPort;      /* graphics port for printing */
   QDProcs  gProcs;     /* procedures for printing in the graphics port */
   long     lGParam1;   /* reserved */
   long     lGParam2;   /* reserved */
   long     lGParam3;   /* reserved */
   long     lGParam4;   /* reserved */
   Boolean  fOurPtr;    /* reserved */
   Boolean  fOurBits;   /* reserved */
};
typedef struct TPrPort TPrPort;
typedef TPrPort *TPPrPort;

enum {feedCut,feedFanfold,feedMechCut,feedOther};
typedef unsigned char TFeed;

enum {scanTB,scanBT,scanLR,scanRL};
typedef unsigned char TScan;

typedef Rect *TPRect;
```

```
typedef pascal void (*PrIdleProcPtr)(void);

typedef pascal void (*PItemProcPtr)(DialogPtr theDialog, short item);

/* structures used by PrGeneral */

struct TGnlData {
   short iOpCode;    /* opcode passed to PrGeneral */
   short iError;     /* result code returned by PrGeneral */
   long  lReserved;  /* more fields here depending on call */
};
typedef struct TGnlData TGnlData;

struct TGetRslBlk {        /* get-resolution record */
   short    iOpCode;       /* the getRslDataOp opcode */
   short    iError;        /* result code returned by PrGeneral */
   long     lReserved;     /* reserved */
   short    iRgType;       /* printer driver version number */
   TRslRg   xRslRg;        /* x-direction resolution range */
   TRslRg   yRslRg;        /* y-direction resolution range */
   short    iRslRecCnt;    /* number of resolution records */
   TRslRec  rgRslRec[27];  /* array of resolution records */
};
typedef struct TGetRslBlk TGetRslBlk;

struct TRslRg {
   short    iMin;          /* minimum resolution supported */
   short    iMax;          /* maximum resolution supported */
};
typedef struct TRslRg TRslRg;

struct TRslRec {
   short    iXRsl;         /* discrete resolution, x direction */
   short    iYRsl;         /* discrete resolution, y direction */
};
typedef struct TRslRec TRslRec;
```

```
struct TSetRslBlk {        /* set-resolution record */
   short    iOpCode;       /* the setRslOp opcode */
   short    iError;        /* result code returned by PrGeneral */
   long     lReserved;     /* reserved */
   THPrint  hPrint;        /* handle to the current TPrint record */
   short    iXRsl;         /* x-direction resolution you want */
   short    iYRsl;         /* y-direction resolution you want */
};
typedef struct TSetRslBlk TSetRslBlk;

struct TDftBitsBlk {       /* draft bits record */
   short    iOpCode;       /* draftBitsOp or noDraftBitsOp opcode */
   short    iError;        /* result code returned by PrGeneral */
   long     lReserved;     /* reserved */
   THPrint  hPrint;        /* handle to the current TPrint record */
};
typedef struct TDftBitsBlk TDftBitsBlk;

struct TGetRotnBlk {       /* page orientation record */
   short    iOpCode;       /* the getRotnOp opcode */
   short    iError;        /* result code returned by PrGeneral */
   long     lReserved;     /* reserved */
   THPrint  hPrint;        /* handle to current TPrint record */
   Boolean  fLandscape;    /* TRUE if user selected landscape printing */
   char     bXtra;         /* reserved */
};
typedef struct TGetRotnBlk TGetRotnBlk;
```

Printing Manager Functions

Opening and Closing the Printing Manager

```
pascal void PrOpen          (void);
pascal void PrClose         (void);
```

Initializing and Validating TPrint Records

```
pascal void PrintDefault    (THPrint hPrint);
pascal Boolean PrValidate   (THPrint hPrint);
```

Displaying and Customizing the Print Dialog Boxes

```
pascal Boolean PrStlDialog  (THPrint hPrint);
pascal Boolean PrJobDialog  (THPrint hPrint);
pascal Boolean PrDlgMain    (THPrint hPrint, PDlgInitProcPtr pDlgInit);
pascal TPPrDlg PrStlInit    (THPrint hPrint);
pascal TPPrDlg PrJobInit    (THPrint hPrint);
pascal void PrJobMerge      (THPrint hPrintSrc, THPrint hPrintDst);
```

Printing a Document

```
pascal TPPrPort PrOpenDoc   (THPrint hPrint, TPPrPort pPrPort, Ptr pIOBuf);
pascal void PrCloseDoc      (TPPrPort pPrPort);
pascal void PrOpenPage      (TPPrPort pPrPort, TPRect pPageFrame);
pascal void PrClosePage     (TPPrPort pPrPort);
pascal void PrPicFile       (THPrint hPrint, TPPrPort pPrPort, Ptr pIOBuf,
                             Ptr pDevBuf, TPrStatus *prStatus);
```

Optimizing Printing

```
pascal void PrGeneral       (Ptr pData);
```

Handling Printing Errors

```
pascal short PrError        (void);
pascal void PrSetError      (short iErr);
```

Low-Level Functions

```
pascal short PrDrvrVers     (void);
pascal void PrDrvrOpen      (void);
pascal void PrDrvrClose     (void);
pascal Handle PrDrvrDCE     (void);
pascal void PrCtlCall       (short iWhichCtl, long lParam1,
                             long lParam2, long lParam3);
```

Application-Defined Functions

```
pascal void MyDoPrintIdle   (void);
pascal TPPrDlg MyPrDialogAppend
                            (THPrint hPrint);
```

Assembly-Language Summary

Data Structures

TPrint Data Structure

0	`iPrVersion`	word	printer driver version that initialized this record
2	`prInfo`	14 bytes	`TPrInfo` data structure with resolution of device and page rectangle
16	`rPaper`	8 bytes	paper rectangle
24	`prStl`	8 bytes	`TPrStl` data structure with printer driver number and feed type
32	`prInfoPT`	14 bytes	`TPrInfo` data structure; reserved
46	`prXInfo`	16 bytes	`TPrXInfo` data structure; reserved
62	`prJob`	20 bytes	`TPrJob` data structure with printing information from the job dialog box
82	`printX`	40 bytes	reserved

TPrInfo Data Structure

0	`iDev`	6 bytes	first word is reserved; second word contains vertical resolution of printer, in dpi; third word contains horizontal resolution of printer, in dpi
6	`rPage`	8 bytes	the page rectangle

TPrJob Data Structure

0	`iFstPage`	word	first page of page range
2	`iLstPage`	word	last page of page range
4	`iCopies`	word	number of copies
6	`bJDocLoop`	1 byte	printing method: draft or deferred
7	`fFromApp`	1 byte	reserved
8	`pIdleProc`	long	pointer to an idle procedure
12	`pFileName`	long	spool filename: `NIL` for default
16	`iFileVol`	word	spool file volume: set to 0 initially
18	`bFileVers`	1 byte	spool file version: set to 0 initially

TPrStl Data Structure

0	`wDev`	word	device number of the current printer (in the high-order byte of this field); the low-order byte of this field is reserved
2	`iPageV`	word	reserved
4	`iPageH`	word	reserved
7	`feed`	1 byte	paper feed type

TPrStatus Data Structure

0	`iTotPages`	word	total pages in print file
2	`iCurPage`	word	current page number
4	`iTotCopies`	word	total copies requested
6	`iCurCopy`	word	current copy number
8	`iTotBands`	word	reserved
10	`iCurBand`	word	reserved
12	`fPgDirty`	1 byte	`TRUE` if current page has been written to
13	`fImaging`	1 byte	reserved
14	`hPrint`	long	handle to the active `TPrint` data structure
18	`pPrPort`	long	pointer to the active printing graphics port
22	`hPic`	long	handle to the active picture

TPrDlg Data Structure

0	`dlg`	168 bytes	a dialog record
168	`pFltrProc`	long	pointer to event filter
172	`pItemProc`	long	pointer to item-handling function
176	`hPrintUsr`	long	handle to `TPrint` data structure
180	`fDoIt`	1 byte	`TRUE` means user clicked OK
181	`fDone`	1 byte	`TRUE` means user clicked OK or Cancel
182	`lUser1`	long	storage for your application
186	`lUser2`	long	storage for your application
190	`lUser3`	long	storage for your application
194	`lUser4`	long	storage for your application

TPrPort Data Structure

0	`gPort`	178 bytes	graphics port and procedures for printing

Trap Macros

Trap Macros Requiring Routine Selectors

_PrGlue

Selector	Routine
$C8000000	PrOpen
$D0000000	PrClose
$20040480	PrintDefault
$52040498	PrValidate
$2A040484	PrStlDialog
$32040488	PrJobDialog
$4A040894	PrDlgMain
$3C04040C	PrStlInit
$44040410	PrJobInit
$5804089C	PrJobMerge
$04000C00	PrOpenDoc
$08000484	PrCloseDoc
$10000808	PrOpenPage
$1800040C	PrClosePage
$60051480	PrPicFile
$BA000000	PrError
$C0000200	PrSetError
$70070480	PrGeneral
$94000000	PrDrvrDCE
$9A000000	PrDrvrVers
$80000000	PrDrvrOpen
$88000000	PrDrvrClose
$A0000E00	PrCtlCall

Global Variable

PrintErr Printing error.

Result Codes

iPrAbort	128	Application or user requested cancel
opNotImpl	2	Requested PrGeneral opcode not implemented in the current printer driver
noSuchRsl	1	Resolution requested with the PrGeneral procedure is not supported
noErr	0	No error
iPrSavPFil	–1	Problem saving print file
controlErr	–17	Unimplemented control instructions; the Device Manager returns this result code
iIOAbort	–27	I/O error
iMemFullErr	–108	There is not enough room in the heap zone
resNotFound	–192	The current printer driver does not support PrGeneral; you should clear this error with a call to PrSetError with a parameter value of 0; otherwise, PrError might still contain this error next time you check it
PAPNoCCBs	–4096	There are no free connect control blocks (CCBs) available
PAPBadRefnum	–4097	Bad connection reference number
PAPActive	–4098	The request is already active
PAPTooBig	–4099	The write request is too big
PAPConnClosed	–4100	The connection is closed
PAPNoPrinter	–4101	The printer is not found, is closed, or is not selected
	–8131	Printer not responding
manualFeedTOErr	–8132	A timeout occurred (that is, no communication has occurred with the printer for two minutes); this is usually caused by an extremely long imaging time or a dropped connection
generalPSErr	–8133	A PostScript error occurred during transmission of data to the printer; this is most often caused by a bug in the application-supplied PostScript code
zoomRangeErr	–8160	The print image enlarged by the user with the Page Setup dialog box overflows the available page resolution
errBadFontKeyType	–8976	Font found in printer is not Type 1, TrueType, or bitmapped font
errPSStateUnderflow	–8977	PostScript stack underflow while restoring graphics state
errNoPattern	–8978	The pixel pattern could not be found and could not be built
errBadConverterID	–8979	The 'PDEF' converter doesn't exist
errNoPagesSpooled	–8980	Application called PrOpenDoc and PrCloseDoc without calling PrOpenPage and PrClosePage in between
errNullColorInfo	–8981	The getColor function called with null GetColorInfo handle
errPSFileNameNull	–8982	The filename pointer for the spool file is null
errSpoolFolderIsAFile	–8983	The spool folder is a file instead of a folder

`errBadConverterIndex`	–8984	When saving a spool file to disk, the value `fileTypeIndex` field had no matching entry in the driver
`errDidNotDownloadFont`	–8985	A PostScript outline could not be found for a PostScript font, and there is no associated `'sfnt'` resource
`errBitmapFontMissing`	–8986	Unable to build bitmap for font
`errPSFileName`	–8987	PostScript file isn't named
`errCouldNotMakeNumberedFilename`	–8989	Could not make a unique filename for the spool file
`errBadSpoolFileVersion`	–8990	Bad version number in header of spool file
`errNoProcSetRes`	–8991	The resource describing needed procedure sets is unavailable for the PostScript prolog
`errInLineTimeout`	–8993	The printer is not responding
`errUnknownPSLevel`	–8994	The PostScript level has an unknown value
`errFontNotFound`	–8995	Font query reply didn't match any fonts in list of PostScript names
`errSizeListBad`	–8996	The size list contained an entry that could not be reconciled with the typeface list
`errFaceListBad`	–8997	Entry could not be found in typeface list
`errNotAKey`	–8998	Key for desired font number and style could not be found in font table

Appendixes

APPENDIX A

Picture Opcodes

Contents

Version and Header Opcodes A-3
Picture Opcode Data Types A-4
Opcodes in Pictures A-5
A Sample Extended Version 2 Picture A-22
A Sample Version 2 Picture A-24
A Sample Version 1 Picture A-25

This appendix describes picture opcodes, which are numbers used by the `DrawPicture` procedure to determine what object to draw or what mode to change for subsequent drawing. Your application generally should not read or write picture opcodes directly but should instead use QuickDraw routines (described in the chapter "Pictures" in this book) for generating and processing the opcodes. Picture opcodes are listed here for your application's debugging purposes.

The `Picture` record (described in the chapter "Pictures") begins with a `picSize` field and a `picFrame` field, followed by a variable amount of picture definition data in the form of opcodes. The first opcode in any picture must be the version opcode, followed by the version number of the picture.

Version and Header Opcodes

In a picture created in extended version 2 or version 2 format, the first opcode is the 2-byte `VersionOp` opcode: $0011. This is followed by the 2-byte `Version` opcode: $02FF. With system software version 4.1 or later, the `Version` opcode identifies the picture as an extended version 2 or a version 2 picture, and all subsequent opcodes are read as words (which are word-aligned within the picture). In versions of system software that precede version 4.1, the $02 is read as the version number, then the $FF is read and interpreted as the end-of-picture opcode—for this reason, `DrawPicture` on a pre-4.1 system terminates without drawing any part of an extended version 2 or version 2 picture.

The 2-byte `HeaderOp` opcode ($0C00) follows the `Version` opcode in an extended version 2 or version 2 format picture. The next 24 bytes contain header information. The value of the 2-byte `version` opcode that follows the `HeaderOp` opcode indicates whether the picture is an extended version 2 picture or a version 2 picture: the `Version` opcode has a value of –2 for an extended version 2 picture and a value of –1 for a version 2 picture. The rest of the header for an extended version 2 picture contains resolution information; the rest of the header for a version 2 picture specifies a fixed-point bounding box.

Opcodes that perform drawing commands follow the header information.
The `OpEndPic` opcode ($00FF) signals the end of the picture for an extended version 2 picture or a version 2 picture.

For an example of the version and header opcodes in a decompiled extended version 2 picture, see Listing A-5 on page A-23. For an example of the version and header opcodes in a decompiled version 2 picture, see Listing A-6 on page A-24.

In a version 1 picture, the `VersionOp` opcode has a value of $11, which is followed by a value of $01. For a version 1 picture, QuickDraw parses the remaining drawing opcodes 1 byte at a time; there is no header information in a version 1 picture. An end-of-picture byte ($FF) after the last opcode or data byte in the file signals the end of the picture.

For an example of the version opcodes in a disassembled version 1 picture, see Listing A-7 on page A-25.

Picture Opcode Data Types

The picture opcodes use the data types that are summarized in Table A-1.

Table A-1 Data types for picture opcodes

Data type	Size
—128..127	1 byte (signed)
0..255	1 byte
Fixed	4 bytes
Integer	2 bytes
Long	4 bytes
Mode	2 bytes
Opcode	2 bytes
Pattern	8 bytes
Point	4 bytes
Poly	10+ bytes
Rect	8 bytes (top, left, bottom, right: integer)
Rgn	10+ bytes
RowBytes	2 bytes (always an even quantity)

In addition, some picture opcode types, such as `BkPixPat`, may use the `PixMap`, `ColorTable`, and `PixData` data types, which makes the length of these opcodes quite variable. The `PixMap` record and `ColorTable` record are described in the chapter "Color QuickDraw" in this book. The following pseudocode describes the `PixData` data type:

```
PixData: {pseudocode describing the PixData data type}
IF rowBytes < 8 THEN
   data is unpacked;
   data size = rowBytes*(bounds.bottom-bounds.top);
IF rowBytes >= 8 THEN
   data is packed;
   image contains (bounds.bottom-bounds.top) packed scanlines;
```

```
   packed scanlines are produced by the PackBits routine;
   each scanline consists of [byteCount] [data];
   IF rowBytes > 250 THEN
      byteCount is a word;
   ELSE
      byteCount is a byte.
END;
```

Opcodes in Pictures

Pictures created with the `OpenPicture` function in a color graphics port use the picture opcodes of the version 2 format. Pictures created with the `OpenCPicture` function use the opcodes of the extended version 2 format. The inclusion of resolution information in the header differentiates the extended version 2 format from the version 2 picture format. The extended version 2 and version 2 formats share the same opcodes, which are listed in Table A-2. The length of the data that follows each 2-byte opcode is listed in this table.

Pictures created with the `OpenPicture` function in a basic graphics port use the opcodes of the version 1 format, which are listed in Table A-3 on page A-18.

The unused opcodes found throughout Table A-2 and Table A-3 are reserved for Apple use. If these opcodes are encountered in pictures, they and their reserved data bytes can simply be skipped. By default, QuickDraw reads and then ignores these opcodes. Because opcodes must be word-aligned in version 2 and extended version 2 pictures, a byte of 0 (zero) data is added after odd-size data.

Note
For opcodes $0100–$7FFF, the amount of data for opcode *$nnXX* = 2 times *nn* bytes. ◆

Table A-2 Opcodes for extended version 2 and version 2 pictures

Opcode	Name	Description	Size (in bytes) of additional data
$0000	`NOP`	No operation	0
$0001	`Clip`	Clipping region	Region size
$0002	`BkPat`	Background pattern	8
$0003	`TxFont`	Font number for text (`Integer`)	2
$0004	`TxFace`	Text's font style (`0..255`)	1
$0005	`TxMode`	Source mode (`Integer`)	2

continued

Table A-2 Opcodes for extended version 2 and version 2 pictures (continued)

Opcode	Name	Description	Size (in bytes) of additional data
$0006	`SpExtra`	Extra space (`Fixed`)	4
$0007	`PnSize`	Pen size (`Point`)	4
$0008	`PnMode`	Pen mode (`Integer`)	2
$0009	`PnPat`	Pen pattern	8
$000A	`FillPat`	Fill pattern	8
$000B	`OvSize`	Oval size (`Point`)	4
$000C	`Origin`	`dh`, `dv` (Integer)	4
$000D	`TxSize`	Text size (`Integer`)	2
$000E	`FgColor`	Foreground color (`Long`)	4
$000F	`BkColor`	Background color (`Long`)	4
$0010	`TxRatio`	Numerator (`Point`), denominator (`Point`)	8
$0011	`VersionOp`	Version (`0..255`)	1
$0012	`BkPixPat`	Background pixel pattern	Variable; see Listing A-1 on page A-17
$0013	`PnPixPat`	Pen pixel pattern	Variable; see Listing A-1 on page A-17
$0014	`FillPixPat`	Fill pixel pattern	Variable; see Listing A-1 on page A-17
$0015	`PnLocHFrac`	Fractional pen position (`Integer`—low word of `Fixed`); if value is not 0.5, pen position is always set to the picture before each text-drawing operation.	2
$0016	`ChExtra`	Added width for nonspace characters (`Integer`)	2
$0017	Reserved for Apple use		Not determined
$0018	Reserved for Apple use		Not determined
$0019	Reserved for Apple use		Not determined
$001A	`RGBFgCol`	Foreground color (`RGBColor`)	6
$001B	`RGBBkCol`	Background color (`RGBColor`)	6

Table A-2 Opcodes for extended version 2 and version 2 pictures (continued)

Opcode	Name	Description	Size (in bytes) of additional data
$001C	`HiliteMode`	Highlight mode flag: no data; this opcode is sent before a drawing operation that uses the highlight mode	0
$001D	`HiliteColor`	Highlight color (`RGBColor`)	6
$001E	`DefHilite`	Use default highlight color; no data; set highlight to default (from low memory)	0
$001F	`OpColor`	Opcolor (`RGBColor`)	6
$0020	`Line`	`pnLoc` (`Point`), `newPt` (`Point`)	8
$0021	`LineFrom`	`newPt` (`Point`)	4
$0022	`ShortLine`	`pnLoc` (`Point`), `dh` (`–128..127`), `dv` (`–128..127`)	6
$0023	`ShortLineFrom`	`dh` (`–128..127`), `dv` (`–128..127`)	2
$0024	Reserved for Apple use	Data length (`Integer`), data	2 + data length
$0025	Reserved for Apple use	Data length (`Integer`), data	2 + data length
$0026	Reserved for Apple use	Data length (`Integer`), data	2 + data length
$0027	Reserved for Apple use	Data length (`Integer`), data	2 + data length
$0028	`LongText`	`txLoc` (`Point`), count (`0..255`), text	5 + text
$0029	`DHText`	`dh` (`0..255`), count (`0..255`), text	2 + text
$002A	`DVText`	`dv` (`0..255`), count (`0..255`), text	2 + text
$002B	`DHDVText`	`dh` (`0..255`), `dv` (`0..255`), count (`0..255`), text	3 + text
$002C	`fontName`	Data length (`Integer`), old font ID (`Integer`), name length (`0..255`), font name*	5 + name length
$002D	`lineJustify`	Operand data length (`Integer`), intercharacter spacing (`Fixed`), total extra space for justification (`Fixed`)†	10
$002E	`glyphState`	Data length (`word`), followed by these 1-byte Boolean values: outline preferred, preserve glyph, fractional widths, scaling disabled	8

continued

Table A-2 Opcodes for extended version 2 and version 2 pictures (continued)

Opcode	Name	Description	Size (in bytes) of additional data
$002F	Reserved for Apple use	Data length (`Integer`), data	2 + data length
$0030	`frameRect`	Rectangle (`Rect`)	8
$0031	`paintRect`	Rectangle (`Rect`)	8
$0032	`eraseRect`	Rectangle (`Rect`)	8
$0033	`invertRect`	Rectangle (`Rect`)	8
$0034	`fillRect`	Rectangle (`Rect`)	8
$0035	Reserved for Apple use	8 bytes of data	8
$0036	Reserved for Apple use	8 bytes of data	8
$0037	Reserved for Apple use	8 bytes of data	8
$0038	`frameSameRect`	Rectangle (`Rect`)	0
$0039	`paintSameRect`	Rectangle (`Rect`)	0
$003A	`eraseSameRect`	Rectangle (`Rect`)	0
$003B	`invertSameRect`	Rectangle (`Rect`)	0
$003C	`fillSameRect`	Rectangle (`Rect`)	0
$003D	Reserved for Apple use		0
$003E	Reserved for Apple use		0
$003F	Reserved for Apple use		0
$0040	`frameRRect`	Rectangle (`Rect`)‡	8
$0041	`paintRRect`	Rectangle (`Rect`)‡	8
$0042	`eraseRRect`	Rectangle (`Rect`)‡	8
$0043	`invertRRect`	Rectangle (`Rect`)‡	8
$0044	`fillRRect`	Rectangle (`Rect`)‡	8
$0045	Reserved for Apple use	8 bytes of data	8
$0046	Reserved for Apple use	8 bytes of data	8
$0047	Reserved for Apple use	8 bytes of data	8
$0048	`frameSameRRect`	Rectangle (`Rect`)	0
$0049	`paintSameRRect`	Rectangle (`Rect`)	0
$004A	`eraseSameRRect`	Rectangle (`Rect`)	0
$004B	`invertSameRRect`	Rectangle (`Rect`)	0
$004C	`fillSameRRect`	Rectangle (`Rect`)	0

Table A-2 Opcodes for extended version 2 and version 2 pictures (continued)

Opcode	Name	Description	Size (in bytes) of additional data
$004D	Reserved for Apple use		0
$004E	Reserved for Apple use		0
$004F	Reserved for Apple use		0
$0050	`frameOval`	Rectangle (`Rect`)	8
$0051	`paintOval`	Rectangle (`Rect`)	8
$0052	`eraseOval`	Rectangle (`Rect`)	8
$0053	`invertOval`	Rectangle (`Rect`)	8
$0054	`fillOval`	Rectangle (`Rect`)	8
$0055	Reserved for Apple use	8 bytes of data	8
$0056	Reserved for Apple use	8 bytes of data	8
$0057	Reserved for Apple use	8 bytes of data	8
$0058	`frameSameOval`	Rectangle (`Rect`)	0
$0059	`paintSameOval`	Rectangle (`Rect`)	0
$005A	`eraseSameOval`	Rectangle (`Rect`)	0
$005B	`invertSameOval`	Rectangle (`Rect`)	0
$005C	`fillSameOval`	Rectangle (`Rect`)	0
$005D	Reserved for Apple use		0
$005E	Reserved for Apple use		0
$005F	Reserved for Apple use		0
$0060	`frameArc`	Rectangle (`Rect`), `startAngle`, `arcAngle`	12
$0061	`paintArc`	Rectangle (`Rect`), `startAngle`, `arcAngle`	12
$0062	`eraseArc`	Rectangle (`Rect`), `startAngle`, `arcAngle`	12
$0063	`invertArc`	Rectangle (`Rect`), `startAngle`, `arcAngle`	12
$0064	`fillArc`	Rectangle (`Rect`), `startAngle`, `arcAngle`	12
$0065	Reserved for Apple use	12 bytes of data	12
$0066	Reserved for Apple use	12 bytes of data	12

continued

Table A-2 Opcodes for extended version 2 and version 2 pictures (continued)

Opcode	Name	Description	Size (in bytes) of additional data
$0067	Reserved for Apple use	12 bytes of data	12
$0068	`frameSameArc`	Rectangle (`Rect`)	4
$0069	`paintSameArc`	Rectangle (`Rect`)	4
$006A	`eraseSameArc`	Rectangle (`Rect`)	4
$006B	`invertSameArc`	Rectangle (`Rect`)	4
$006C	`fillSameArc`	Rectangle (`Rect`)	4
$006D	Reserved for Apple use	4 bytes of data	4
$006E	Reserved for Apple use	4 bytes of data	4
$006F	Reserved for Apple use	4 bytes of data	4
$0070	`framePoly`	Polygon (`Poly`)	Polygon size
$0071	`paintPoly`	Polygon (`Poly`)	Polygon size
$0072	`erasePoly`	Polygon (`Poly`)	Polygon size
$0073	`invertPoly`	Polygon (`Poly`)	Polygon size
$0074	`fillPoly`	Polygon (`Poly`)	Polygon size
$0075	Reserved for Apple use	Polygon (`Poly`)	Polygon size
$0076	Reserved for Apple use	Polygon (`Poly`)	Polygon size
$0077	Reserved for Apple use	Polygon (`Poly`)	Polygon size
$0078	`frameSamePoly`	(Not yet implemented)	0
$0079	`paintSamePoly`	(Not yet implemented)	0
$007A	`eraseSamePoly`	(Not yet implemented)	0
$007B	`invertSamePoly`	(Not yet implemented)	0
$007C	`fillSamePoly`	(Not yet implemented)	0
$007D	Reserved for Apple use		0
$007E	Reserved for Apple use		0
$007F	Reserved for Apple use		0
$0080	`frameRgn`	Region (`Rgn`)	Region size
$0081	`paintRgn`	Region (`Rgn`)	Region size
$0082	`eraseRgn`	Region (`Rgn`)	Region size
$0083	`invertRgn`	Region (`Rgn`)	Region size

Table A-2 Opcodes for extended version 2 and version 2 pictures (continued)

Opcode	Name	Description	Size (in bytes) of additional data
$0084	`fillRgn`	Region (`Rgn`)	Region size
$0085	Reserved for Apple use	Region (`Rgn`)	Region size
$0086	Reserved for Apple use	Region (`Rgn`)	Region size
$0087	Reserved for Apple use	Region (`Rgn`)	Region size
$0088	`frameSameRgn`	(Not yet implemented)	0
$0089	`paintSameRgn`	(Not yet implemented)	0
$008A	`eraseSameRgn`	(Not yet implemented)	0
$008B	`invertSameRgn`	(Not yet implemented)	0
$008C	`fillSameRgn`	(Not yet implemented)	0
$008D	Reserved for Apple use		0
$008E	Reserved for Apple use		0
$008F	Reserved for Apple use		0
$0090	`BitsRect`	`CopyBits` with clipped rectangle	Variable§¶; see Listing A-2 on page A-17
$0091	`BitsRgn`	`CopyBits` with clipped region	Variable§¶; see Listing A-3 on page A-18
$0092	Reserved for Apple use	Data length (`Integer`), data	2 + data length
$0093	Reserved for Apple use	Data length (`Integer`), data	2 + data length
$0094	Reserved for Apple use	Data length (`Integer`), data	2 + data length
$0095	Reserved for Apple use	Data length (`Integer`), data	2 + data length
$0096	Reserved for Apple use	Data length (`Integer`), data	2 + data length
$0097	Reserved for Apple use	Data length (`Integer`), data	2 + data length
$0098	`PackBitsRect`	Packed `CopyBits` with clipped rectangle	Variable§; see Listing A-2 on page A-17
$0099	`PackBitsRgn`	Packed `CopyBits` with clipped rectangle	Variable§; see Listing A-3 on page A-18
$009A	`DirectBitsRect`	`PixMap`, `srcRect`, `dstRect`, `mode` (`Integer`), `PixData`	Variable

continued

Table A-2 Opcodes for extended version 2 and version 2 pictures (continued)

Opcode	Name	Description	Size (in bytes) of additional data
$009B	`DirectBitsRgn`	`PixMap`, `srcRect`, `dstRect`, `mode` (`Integer`), `maskRgn`, `PixData`	Variable
$009C	Reserved for Apple use	Data length (`Integer`), data	2 + data length
$009D	Reserved for Apple use	Data length (`Integer`), data	2 + data length
$009E	Reserved for Apple use	Data length (`Integer`), data	2 + data length
$009F	Reserved for Apple use	Data length (`Integer`), data	2 + data length
$00A0	`ShortComment`	Kind (`Integer`)	2
$00A1	`LongComment`	Kind (`Integer`), size (`Integer`), data	4 + data
$00A2	Reserved for Apple use	Data length (`Integer`), data	2 + data length
.	.	.	.
.	.	.	.
.	.	.	.
$00AF	Reserved for Apple use	Data length (`Integer`), data	2 + data length
$00B0	Reserved for Apple use		0
.	.	.	.
.	.	.	.
.	.	.	.
$00CF	Reserved for Apple use		0
$00D0	Reserved for Apple use	Data length (`Long`), data	4 + data length
.	.	.	.
.	.	.	.
.	.	.	.
$00FE	Reserved for Apple use	Data length (`Long`), data	4 + data length
$00FF	`OpEndPic`	End of picture	2
$0100	Reserved for Apple use	2 bytes of data	2
.	.	.	.
.	.	.	.
.	.	.	.

Table A-2 Opcodes for extended version 2 and version 2 pictures (continued)

Opcode	Name	Description	Size (in bytes) of additional data
$01FF	Reserved for Apple use	2 bytes of data	2
$0200	Reserved for Apple use	4 bytes of data	4
$02FF	`Version`	Version number of picture	2
.	.	.	.
.	.	.	.
.	.	.	.
$0BFF	Reserved for Apple use	22 bytes of data	22
$0C00	`HeaderOp`	For extended version 2: version (`Integer`), reserved (`Integer`), `hRes`, `vRes` (`Fixed`), `srcRect`, reserved (`Long`); for version 2: opcode	24
$0C01	Reserved for Apple use	24 bytes of data	24
.	.	.	.
.	.	.	.
.	.	.	.
$7F00	Reserved for Apple use	254 bytes of data	254
.	.	.	.
.	.	.	.
.	.	.	.
$7FFF	Reserved for Apple use	254 bytes of data	254
$8000	Reserved for Apple use		0
.	.	.	.
.	.	.	.
.	.	.	.
$80FF	Reserved for Apple use		0
$8100	Reserved for Apple use	Data length (`Long`), data	4 + data length
.	.	.	.
.	.	.	.
.	.	.	.

continued

Table A-2 Opcodes for extended version 2 and version 2 pictures (continued)

Opcode	Name	Description	Size (in bytes) of additional data
$8200	`CompressedQuickTime`	Data length (`Long`), data (private to QuickTime)	4 + data length
$8201	`UncompressedQuickTime`	Data length (`Long`), data (private to QuickTime)	4 + data length
$FFFF	Reserved for Apple use	Data length (`Long`), data	4 + data length

* The font name information begins with a word containing the field's data length, followed by a word containing the old font ID, a byte containing the length of the font name, and then the font name itself.

You can extract font names, IDs, and other information from a picture by using the routines described in the chapter "Pictures" in this book.

† For opcode $002D (`lineJustify`), the line justification information contains the line-layout state of the Script Manager so that it can be restored when the picture is played back. It begins with a word containing the field's data length, which should always be 8 bytes. The operands are two fixed-point values, describing the Script Manager's extra character width value and the total extra width that was added to the style run (each `StdText` call) to perform justification.

For example, if the intercharacter spacing were 1 pixel and the total extra width added were 10 pixels, the following hexadecimal bytes would be generated for the picture:

```
2D 00 08 00 01 00 00 00 0A 00 00
```

In this example, the `$002D` opcode is followed by the length word, 00 08, and then the integer part of the intercharacter spacing, 00 01, its fractional part, 00 00, and then the integer part of the total extra spacing, 00 0A, and its fractional part, 00 00.

‡ For opcodes $0040–$0044: rounded rectangles use the setting of the `OvSize` point (refer to opcode $000B).

§ Four opcodes ($0090, $0091, $0098, $0099) are modifications of version 1 opcodes. The first word following the opcode is `rowBytes`. If the high bit of `rowBytes` is set, then it is a pixel map containing multiple bits per pixel; if it is not set, it is a bitmap containing 1 bit per pixel. In general, the difference between version 2 and version 1 formats is that the pixel map replaces the bitmap, a color table has been added, and `pixData` replaces `bitData`.

¶ For opcodes $0090 (`BitsRect`) and $0091 (`BitsRgn`), the data is unpacked. These opcodes can be used only when `rowBytes` is less than 8.

Opcodes $009A (`DirectBitsRect`) and $009B (`DirectBitsRgn`) define direct-pixel pictures, with pixel maps containing three components that directly specify RGB colors. These opcodes allow your application to cut, paste, and store images with up to 32 bits of color information per pixel.

The `DirectBitsRect` and `DirectBitsRgn` opcodes store the `baseAddr` field of the `PixMap` record in a version 2 picture. For compatibility with existing systems, the `baseAddr` field is set to $000000FF. Black-and-white video devices can display pixel maps that are in pictures. On systems without direct-pixel support, opcodes $009A and $009B read a word from the picture and then skip a word of data. The next opcode

retrieved from the picture is $00FF, which terminates picture playback. (Note that if you play back a picture on a machine without direct-pixel support, it terminates picture parsing.)

The `DirectBitsRect` opcode is followed by this structure:

```
pixMap:     PixMap;
srcRect:    Rect;    {source rectangle}
dstRect:    Rect;    {destination rectangle}
mode:       Mode;    {transfer mode}
pixData:
```

The `DirectBitsRgn` opcode is followed by this structure:

```
pixMap:     PixMap;
srcRect:    Rect;    {source rectangle}
dstRect:    Rect;    {destination rectangle}
mode:       Mode;    {transfer mode}
maskRgn:    Region;  {region for masking}
pixData:
```

In a picture, the `packType` field of a `PixMap` record specifies the manner in which the pixel data was compressed. To facilitate banding of images when memory is short, all data compression is done on a scan-line basis. The following pseudocode describes the pixel data:

```
PixData:
IF packType = 1 (unpacked) OR rowbytes < 8 THEN
   data is unpacked;
   data size = rowBytes * (bounds.bottom - bounds.top);

IF packType = 2 (drop pad byte) THEN
   the high-order pad byte of a 32-bit direct pixel is
   dropped;
   data size = (3/4) * rowBytes *
   (bounds.bottom - bounds.top);

IF packType > 2 (packed) THEN
   image contains (bounds.bottom - bounds.top) packed
   scan lines;
   each scan line consists of [byteCount] [data];
   IF rowBytes > 250 THEN
      byteCount is a word
   ELSE
      it is a byte
```

Here are the currently defined packing types:

Packing type	Meaning
0	Use default packing
1	Use no packing
2	Remove pad byte—supported only for 32-bit pixels (24-bit data)
3	Run length encoding by `pixelSize` chunks, one scan line at a time—supported only for 16-bit pixels
4	Run length encoding one component at a time, one scan line at a time, red component first—supported only for 32-bit pixels (24-bit data)

For future compatibility, other `packType` values skip scan-line data and draw nothing. Since QuickDraw assumes that pixel map data in memory is unpacked regardless of the `packType` field value, you can use `packType` to tell the picture-recording mechanism what packing technique to use on that data. A `packType` value of 0 in memory indicates that the default packing scheme should be used. (Using the default packing scheme is recommended.) Currently, the default `packType` value for a `pixelSize` value of 16 is type 3; for a `pixelSize` value of 32, it is type 4. Regardless of the setting of `packType` at the time of picture recording, the `packType` value actually used to save the image is recorded in the picture.

Since each scan line of packed data is preceded by a byte count, `packSize` is not used and must be 0 for future compatibility.

When the pixel type is direct, `cmpCount * cmpSize` is less than or equal to `pixelSize`. For storing 24-bit data in a 32-bit pixel, set `cmpSize` to 8 and `cmpCount` to 3. If you set `cmpCount` to 4, then the high byte is compressed by packing scheme 4 and stored in the picture.

The `OpenCPicture` function lets your application create a version 2 format picture and include rectangle and resolution information, which is stored in the version 2 picture header. The `OpenCPicture` function is described in the chapter “Pictures.”

The `HeaderOp` information is passed to the `OpenCPicture` function as an `OpenCPicParams` record, which is described in the chapter “Pictures” in this book.

The pseudocode in Listing A-1 illustrates the data for the `BkPixPat`, `PnPixPat`, and `FillPixPat` opcodes.

Listing A-1 Data for the `BkPixPat`, `PnPixPat`, and `FillPixPat` opcodes

```
IF patType = ditherPat
THEN
   PatType:    word;       {pattern type = 2}
   Pat1Data:   Pattern;    {old pattern data}
   RGB:        RGBColor;   {desired RGB for pattern}
ELSE
   PatType:    word;       {pattern type = 1}
   Pat1Data:   Pattern;    {old pattern data}
   PixMap:     PixMap;
   ColorTable: ColorTable;
   PixData:    PixData;
END;
```

The pseudocode in Listing A-2 illustrates the data is stored in the `BitsRect` and `PackBitsRect` opcodes.

Listing A-2 Data for the `BitsRect` and `PackBitsRect` opcodes

```
   PixMap:     PixMap;     {pixel map}
   ColorTable: ColorTable; {ColorTable record}
   srcRect:    Rect;       {source rectangle}
   dstRect:    Rect;       {destination rectangle}
   mode:       Word;       {transfer mode (may include }
                           { new transfer modes)}
   PixData:    PixData;
```

The pseudocode in Listing A-3 illustrates the data is stored in the `BitsRgn` and `PackBitsRgn` opcodes.

Listing A-3 Data for the `BitsRgn` and `PackBitsRgn` opcodes

```
pixMap:     PixMap;
colorTable: ColorTable;
srcRect:    Rect;          {source rectangle}
dstRect:    Rect;          {destination rectangle}
mode:       Word;          {transfer mode (may }
                           { include new modes)}
maskRgn:    Rgn;           {region for masking}
pixData:    PixData;
```

Pictures created with the `OpenPicture` function in a basic graphics port use the opcodes of the version 1 format, as listed in Table A-3. This size of data that follows each opcode is also listed in this table. Version 1 pictures are limited to 32 KB.

Table A-3 Opcodes for version 1 pictures

Opcode	Name	Description	Size (in bytes) of additional data
$00	`NOP`	No operation	0
$01	`ClipRgn`	Clipping region	Region size
$02	`BkPat`	Background pattern	8
$03	`TxFont`	Font number for text (`Integer`)	2
$04	`TxFace`	Text's font style (`0..255`)	1
$05	`TxMode`	Source mode (`Integer`)	2
$06	`SpExtra`	Extra space (`Fixed`)	4
$07	`PnSize`	Pen size (`Point`)	4
$08	`PnMode`	Pen mode (`Integer`)	2
$09	`PnPat`	Pen pattern	8
$0A	`FillPat`	Fill pattern	8
$0B	`OvSize`	Oval size (`Point`)	4
$0C	`Origin`	`dh` (`Integer`), `dv` (`Integer`)	4
$0D	`TxSize`	Text size (`Integer`)	2
$0E	`FgColor`	Foreground color (`Long`)	4
$0F	`BkColor`	Background color (`Long`)	4

Table A-3 Opcodes for version 1 pictures (continued)

Opcode	Name	Description	Size (in bytes) of additional data
$10	`TxRatio`	Numerator (`Point`), denominator (`Point`)	8
$11	`picVersion`	Version (`0..255`)	1
$20	`Line`	`pnLoc` (`Point`), `newPt` (`Point`)	8
$21	`LineFrom`	`newPt` (`Point`)	4
$22	`ShortLine`	`pnLoc` (`Point`), `dh` (`–128..127`), `dv` (`–128..127`)	6
$23	`ShortLineFrom`	`dh` (`–128..127`), `dv` (`–128..127`)	2
$28	`LongText`	`txLoc` (`Point`), count (`0..255`), text	5 + text
$29	`DHText`	`dh` (`0..255`), count (`0..255`), text	2 + text
$2A	`DVText`	`dv` (`0..255`), count (`0..255`), text	2 + text
$2B	`DHDVText`	`dh` (`0..255`), `dv` (`0..255`), count (`0..255`), text	3 + text
$30	`frameRect`	Rectangle (`Rect`)	8
$31	`paintRect`	Rectangle (`Rect`)	8
$32	`eraseRect`	Rectangle (`Rect`)	8
$33	`invertRect`	Rectangle (`Rect`)	8
$34	`fillRect`	Rectangle (`Rect`)	8
$38	`frameSameRect`	Rectangle (`Rect`)	0
$39	`paintSameRect`	Rectangle (`Rect`)	0
$3A	`eraseSameRect`	Rectangle (`Rect`)	0
$3B	`invertSameRect`	Rectangle (`Rect`)	0
$3C	`fillSameRect`	Rectangle (`Rect`)	0
$40	`frameRRect`	Rectangle (`Rect`)*	8
$41	`paintRRect`	Rectangle (`Rect`)*	8
$42	`eraseRRect`	Rectangle (`Rect`)*	8
$43	`invertRRect`	Rectangle (`Rect`)*	8
$44	`fillRRect`	Rectangle (`Rect`)*	8
$48	`frameSameRRect`	Rectangle (`Rect`)	0
$49	`paintSameRRect`	Rectangle (`Rect`)	0
$4A	`eraseSameRRect`	Rectangle (`Rect`)	0

continued

Table A-3 Opcodes for version 1 pictures (continued)

Opcode	Name	Description	Size (in bytes) of additional data
$4B	`invertSameRRect`	Rectangle (`Rect`)	0
$4C	`fillSameRRect`	Rectangle (`Rect`)	0
$50	`frameOval`	Rectangle (`Rect`)	8
$51	`paintOval`	Rectangle (`Rect`)	8
$52	`eraseOval`	Rectangle (`Rect`)	8
$53	`invertOval`	Rectangle (`Rect`)	8
$54	`fillOval`	Rectangle (`Rect`)	8
$58	`frameSameOval`	Rectangle (`Rect`)	0
$59	`paintSameOval`	Rectangle (`Rect`)	0
$5A	`eraseSameOval`	Rectangle (`Rect`)	0
$5B	`invertSameOval`	Rectangle (`Rect`)	0
$5C	`fillSameOval`	Rectangle (`Rect`)	0
$60	`frameArc`	Rectangle (`Rect`), `startAngle`, `arcAngle`	12
$61	`paintArc`	Rectangle (`Rect`), `startAngle`, `arcAngle`	12
$62	`eraseArc`	Rectangle (`Rect`), `startAngle`, `arcAngle`	12
$63	`invertArc`	Rectangle (`Rect`), `startAngle`, `arcAngle`	12
$64	`fillArc`	Rectangle (`Rect`), `startAngle`, `arcAngle`	12
$68	`frameSameArc`	Rectangle (`Rect`)	4
$69	`paintSameArc`	Rectangle (`Rect`)	4
$6A	`eraseSameArc`	Rectangle (`Rect`)	4
$6B	`invertSameArc`	Rectangle (`Rect`)	4
$6C	`fillSameArc`	Rectangle (`Rect`)	4
$70	`framePoly`	Polygon (`Poly`)	Polygon size
$71	`paintPoly`	Polygon (`Poly`)	Polygon size
$72	`erasePoly`	Polygon (`Poly`)	Polygon size
$73	`invertPoly`	Polygon (`Poly`)	Polygon size
$74	`fillPoly`	Polygon (`Poly`)	Polygon size
$78	`frameSamePoly`	(Not yet implemented)	0

Table A-3 Opcodes for version 1 pictures (continued)

Opcode	Name	Description	Size (in bytes) of additional data
$79	`paintSamePoly`	(Not yet implemented)	0
$7A	`eraseSamePoly`	(Not yet implemented)	0
$7B	`invertSamePoly`	(Not yet implemented)	0
$7C	`fillSamePoly`	(Not yet implemented)	0
$80	`frameRgn`	Region (`Rgn`)	Region size
$81	`paintRgn`	Region (`Rgn`)	Region size
$82	`eraseRgn`	Region (`Rgn`)	Region size
$83	`invertRgn`	Region (`Rgn`)	Region size
$84	`fillRgn`	Region (`Rgn`)	Region size
$88	`frameSameRgn`	(Not yet implemented)	0
$89	`paintSameRgn`	(Not yet implemented)	0
$8A	`eraseSameRgn`	(Not yet implemented)	0
$8B	`invertSameRgn`	(Not yet implemented)	0
$8C	`fillSameRgn`	(Not yet implemented)	0
$90	`BitsRect`	`CopyBits` with clipped rectangle	Variable†‡; see Listing A-2 on page A-17
$91	`BitsRgn`	`CopyBits` with clipped region	Variable†‡; see Listing A-3 on page A-18
$98	`PackBitsRect`	Packed `CopyBits` with clipped rectangle	Variable†; see Listing A-2 on page A-17
$99	`PackBitsRgn`	Packed `CopyBits` with clipped rectangle	Variable†; see Listing A-3 on page A-18
$A0	`ShortComment`	Kind (`Integer`)	2
$A1	`LongComment`	Kind (`Integer`), size (`Integer`), data	4 + data
$FF	`EndOfPicture`	End of picture	0

* For opcodes $40–$44: rounded rectangles use the setting of the `OvSize` point (refer to opcode $0B).

† In general, the difference between version 2 and version 1 formats is that the pixel map replaces the bitmap, a color table has been added, and `pixData` replaces `bitData`.

‡ For opcodes $90 (`BitsRect`) and $91 (`BitsRgn`), the data is unpacked. These opcodes can only be used when `rowBytes` is less than 8.

A Sample Extended Version 2 Picture

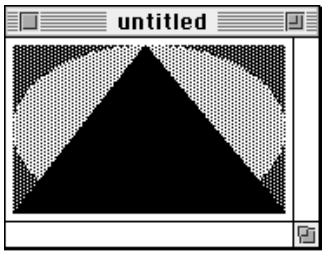
The chapter "Pictures" in this book describes how to use the `OpenCPicture` function to create and display extended version 2 pictures. Listing A-4 illustrates how to use `OpenCPicture`.

Listing A-4 Creating and drawing an extended version 2 picture

```
FUNCTION MyCreateAndDrawPict(pFrame: Rect): PicHandle;
VAR
   myOpenCPicParams: OpenCPicParams;
   myPic:            PicHandle;
   trianglePoly:     PolyHandle;
BEGIN
   WITH myOpenCPicParams DO BEGIN
      srcRect := pFrame;
      hRes := gHRes;       {$00480000 for 72 dpi}
      vRes := gVRes;       {$00480000 for 72 dpi}
      version := - 2;      {always set this field to -2}
      reserved1 := 0;      {this field is unused}
      reserved2 := 0;      {this field is unused}
   END;
   myPic := OpenCPicture(myOpenCPicParams);  {start creating the picture}
   ClipRect(pFrame);                        {always set a valid clip region}
   FillRect(pFrame,dkGray);   {create a dark gray rectangle for background}
   FillOval(pFrame,ltGray);   {overlay the rectangle with a light gray oval}
   trianglePoly := OpenPoly;  {start creating a triangle}
   WITH pFrame DO BEGIN
      MoveTo(left,bottom);
      LineTo((right - left) DIV 2,top);
      LineTo(right,bottom);
      LineTo(left,bottom);
   END;
   ClosePoly;                 {finish the triangle}
   PaintPoly(trianglePoly);   {paint the triangle}
   KillPoly(trianglePoly);    {dispose of the memory for the triangle}
   ClosePicture;              {finish the picture}
   DrawPicture(myPic,pFrame);       {draw the picture}
   MyCreateAndDrawPict := myPic;
END;
```

Figure A-1 shows the picture created by Listing A-4.

Figure A-1 A picture



The QuickDraw drawing commands issued between `OpenCPicture` and the `ClosePicture` procedure in Listing A-4 are saved in memory as a `Picture` record containing a `picSize` field, a `picFrame` field, and an array of picture opcodes; an application can also save this information in a resource of type `'PICT'`. The `DrawPicture` procedure reads these opcodes when drawing the picture.

For debugging purposes, you might find it helpful to examine the opcodes for a picture. Listing A-5 shows the extended version 2 picture in Figure A-1 after it is saved in a `'PICT'` resource and then decompiled with the DeRez decompiler.

Listing A-5 A decompiled extended version 2 picture

```
data 'PICT' (128) {
$"0078"     /* picture size; don't use this value for picture size */
$"0000 0000 006C 00A8"  /* bounding rectangle of picture at 72 dpi */
$"0011"     /* VersionOp opcode; always $0011 for extended version 2 */
$"02FF"     /* Version opcode; always $02FF for extended version 2 */
$"0C00"     /* HeaderOp opcode; always $0C00 for extended version 2 */
            /* next 24 bytes contain header information */
   $"FFFE"  /* version; always -2 for extended version 2 */
   $"0000"  /* reserved */
   $"0048 0000"         /* best horizontal resolution: 72 dpi */
   $"0048 0000"         /* best vertical resolution: 72 dpi */
   $"0002 0002 006E 00AA"  /* optimal source rectangle for 72 dpi horizontal
                              and 72 dpi vertical resolutions */
   $"0000"  /* reserved */
$"001E"     /* DefHilite opcode to use default hilite color */
$"0001"     /* Clip opcode to define clipping region for picture */
   $"000A"  /* region size */
   $"0002 0002 006E 00AA"  /* bounding rectangle for clipping region */
$"000A"     /* FillPat opcode; fill pattern specified in next 8 bytes */
```

```
   $"77DD 77DD 77DD 77DD"  /* fill pattern */
$"0034"     /* fillRect opcode; rectangle specified in next 8 bytes */
   $"0002 0002 006E 00AA"  /* rectangle to fill */
$"000A"     /* FillPat opcode; fill pattern specified in next 8 bytes */
   $"8822 8822 8822 8822"  /* fill pattern */
$"005C"     /* fillSameOval opcode */
$"0008"     /* PnMode opcode */
$  "0008"   /* pen mode data */
$"0071"     /* paintPoly opcode */
   $"001A"  /* size of polygon */
   $"0002 0002 006E 00AA"  /* bounding rectangle for polygon */
   $"006E 0002 0002 0054 006E 00AA 006E 0002"   /* polygon points */
$"00FF"     /* OpEndPic opcode; end of picture */
}
```

A Sample Version 2 Picture

The chapter "Pictures" in this book describes how to use the `OpenPicture` function, which creates version 2 pictures in color graphics ports. Figure A-1 on page A-23 shows a picture created with the `OpenCPicture` function using the code in Listing A-4 on page A-22. If the `OpenPicture` function were used instead of `OpenCPicture`, the same picture would be drawn, but the picture would use picture opcodes for the version 2 format instead of the extended version 2 format. The major difference between formats lies in the header information after the `HeaderOp` opcode.

Listing A-6 shows what happens when the picture in Figure A-1 is created in version 2 format, saved in a `'PICT'` resource, and then decompiled with the DeRez decompiler.

Listing A-6 A decompiled version 2 picture

```
data 'PICT' (129) {
$"0078"     /* picture size; don't use this value for picture size */
$"0002 0002 006E 00AA"  /* bounding rectangle of picture */
$"0011"     /* VersionOp opcode; always $0011 for version 2 */
$"02FF"     /* Version opcode; always $02FF for version 2 */
$"0C00"     /* HeaderOp opcode; always $0C00 for version 2 */
            /* next 24 bytes contain header information */
   $"FFFF FFFF"   /* version; always -1 (long) for version 2 */
   $"0002 0000 0002 0000 00AA 0000 006E 0000"   /* fixed-point bounding
                                                   rectangle for picture */
   $"0000 0000"   /* reserved */
$"001E"     /* DefHilite opcode to use default hilite color */
```

```
$"0001"     /* Clip opcode to define clipping region for picture */
   $"000A"  /* region size */
   $"0002 0002 006E 00AA"  /* bounding rectangle for clipping region */
$"000A"     /* FillPat opcode; fill pattern specifed in next 8 bytes */
   $"77DD 77DD 77DD 77DD"  /* fill pattern */
$"0034"     /* fillRect opcode; rectangle specified in next 8 bytes */
   $"0002 0002 006E 00AA"  /* rectangle to fill */
$"000A"     /* FillPat opcode; fill pattern specified in next 8 bytes */
   $"8822 8822 8822 8822"  /* fill pattern */
$"005C"     /* fillSameOval opcode */
$"0008"     /* PnMode opcode */
$  "0008"   /* pen mode data */
$"0071"     /* paintPoly opcode */
   $"001A"  /* size of polygon */
   $"0002 0002 006E 00AA"  /* bounding rectangle for polygon */
   $"006E 0002 0002 0054 006E 00AA 006E 0002"   /* polygon points */
$"00FF"     /* OpEndPic opcode; end of picture */
}
```

A Sample Version 1 Picture

Pictures created by the `OpenPicture` function on computers without Color QuickDraw, or when the current graphics port is a basic graphics port, are created in version 1 format. The code in Listing A-7 shows what happens when the picture in Figure A-1 on page A-23 is created in version 1 format, saved in a `'PICT'` resource, and then decompiled with the DeRez decompiler.

Listing A-7 A decompiled version 1 picture

```
data 'PICT' (130) {
$"004F"     /* picture size; this value is reliable for version 1 pictures */
$"0002 0002 006E 00AA"  /* bounding rectangle of picture */
$"11"       /* picVersion opcode for version 1 */
   $"01"    /* version number 1 */
$"01"       /* ClipRgn opcode to define clipping region for picture */
   $"000A"  /* region size */
   $"0002 0002 006E 00AA"  /* bounding rectangle for region */
$"0A"       /* FillPat opcode; fill pattern specified in next 8 bytes */
   $"77DD 77DD 77DD 77DD"  /* fill pattern */
$"34"       /* fillRect opcode; rectangle specified in next 8 bytes */
   $"0002 0002 006E 00AA"  /* rectangle to fill */
```

```
$"0A"       /* FillPat opcode; fill pattern specified in next 8 bytes */
   $"8822 8822 8822 8822"  /* fill pattern */
$"5C"       /* fillSameOval opcode */
$"71"       /* paintPoly opcode */
   $"001A"  /* size of polygon */
   $"0002 0002 006E 00AA"  /* bounding rectangle for polygon */
   $"006E 0002 0002 0054 006E 00AA 006E 0002"   /* polygon points */
$"FF"       /* EndOfPicture opcode; end of picture */
}
```

Using Picture Comments for Printing

Contents

About Picture Comments B-3
Maintaining Device Independence B-8
Synchronizing QuickDraw and PostScript Printer Drivers B-10
Using Text Picture Comments B-11
  Disabling and Reenabling Line Layout B-11
  Delimiting Strings B-16
  Rotating Text B-17
Using Graphics Picture Comments B-22
  Drawing Polygons B-23
  Rotating Graphics B-29
Using Line-Drawing Picture Comments B-33
  Drawing Dashed Lines B-33
  Using Fractional Line Widths B-35
Using PostScript Picture Comments B-38
  Calling PostScript Routines Directly B-38
  Optimizing PostScript Printing B-39
Picture Comments to Avoid B-40
Including Constants and Data Types for Picture Comments B-42

This appendix describes the picture comments predefined by Apple Computer, Inc., for its PostScript printers and several of its QuickDraw printers (including the LaserWriter SC, ImageWriter LQ, and StyleWriter printers). This appendix introduces you to the use of picture comments for printing with features that are unavailable with QuickDraw alone.

For most applications, sending QuickDraw's picture-drawing routines to the printer driver is sufficient: the driver either uses QuickDraw or converts QuickDraw routines to PostScript code. See the chapter "Printing Manager" in this book for information about QuickDraw-based printing. For some applications, such as page-layout programs, QuickDraw-based printing may not be sufficient; such applications may rely on printer drivers—such as PostScript printer drivers—to provide features that are not available, or are difficult to achieve, using QuickDraw.

For PostScript printers, one solution is for your application to send PostScript code directly to the printer driver, but this approach requires you to know the PostScript language as well as QuickDraw. If your application requires features (such as rotated text and dashed lines) that are unavailable with QuickDraw, you may instead want to use picture comments to take advantage of these features on capable printers. Created with the QuickDraw procedure `PicComment`, picture comments are data or commands for special processing by output devices such as printer drivers. The `PicComment` procedure is introduced in the chapter "Pictures" in this book and is expanded upon in this appendix.

IMPORTANT
The picture comments supported by Apple printer drivers are described on page B-7. However, it is impossible to determine which picture comments are supported by the current printer driver. ▲

About Picture Comments

Within the drawing code sent to a printer driver after your application uses the `PrOpenPage` procedure, your application can specify picture comments by using the QuickDraw `PicComment` procedure. The `PicComment` procedure allows your application to pass data or commands directly to an output device.

```
PROCEDURE PicComment (kind: Integer; dataSize: Integer;
                      dataHandle: Handle);
```

The `kind` parameter specifies the kind of picture comment, and the `dataSize` parameter specifies the size of the data referred to by the `dataHandle` parameter. (For some picture comments, the values passed in the `dataSize` and `dataHandle` parameters should be 0 and `NIL`, respectively.)

You typically use a picture comment to give your application and an output device additional control over the rendering of images. A number of picture comments have been given special definitions by various printer drivers. When a printer driver encounters one of these comments, it interprets the comment as an appropriate drawing operation. A PostScript printer driver, for example, may convert a picture comment into PostScript code.

By including picture comments in your code that draws into a printing graphics port, your application can rotate text and graphics, smooth polygons, draw hairlines, create dashed lines, and pass PostScript code directly to the printer driver. (For information about the PostScript language, see the *PostScript Language Reference Manual*, second edition, published by Addison-Wesley.)

Picture comments were initially designed to allow applications to share data in the form of QuickDraw pictures (as described in the chapter "Pictures" in this book). With the advent of the PostScript LaserWriter printer, the use of picture comments was extended to allow applications to more easily take advantage of various PostScript features unavailable with QuickDraw.

However, you do not need to create a QuickDraw picture to use picture comments for printing. When your application calls the Printing Manager procedure `PrOpenPage`, the printer driver collects your drawing operations after they are handled by the low-level drawing routines contained in the `QDProcs` record for the printing graphics port. As explained in the chapter "QuickDraw Drawing" in this book, the default low-level procedure specified by QuickDraw in the `commentProc` field of the `QDProcs` record is the `StdComment` procedure, which simply ignores picture comments. However, a printer driver can replace the `StdComment` procedure with its own routine for handling picture comments.

▲ **WARNING**
As described in the chapter "Pictures" in this book, do not call the `OpenCPicture` or `OpenPicture` function between calls to `PrOpenPage` and `PrClosePage`. ▲

When you use the `PicComment` procedure after calling `PrOpenPage` and before calling `PrClosePage`, the printer driver either ignores the picture comment passed to `PicComment` or collects the results of its drawing operations, depending on whether the printer driver has installed its own low-level drawing routine that handles the picture comment.

Although the `PicComment` procedure is available on all Macintosh computers, the availability of the drawing operations that you can implement with picture comments depends on the driver for the current printer. The inability to determine which picture comments are supported by the current printer driver means that if you use picture comments to perform drawing operations not supported by QuickDraw, you must also provide for printing on QuickDraw-only printers.

This requires your application to maintain separate code branches: for example, one that takes advantage of the picture comment handling of a PostScript printer driver, and another for a printer driver that supports only QuickDraw. Furthermore, you must hide the code that takes advantage of PostScript printer drivers from QuickDraw-based drivers, and you must hide from PostScript drivers the code that uses QuickDraw-based approximations of these drawing operations. Your application's printed output will necessarily differ depending on the driver for the current printer.

Table B-1 lists picture comments defined for various printer drivers produced by Apple and used by third-party producers of various other printer drivers. For each picture comment, this table shows the name of the picture comment that you specify in the `kind` parameter of the `PicComment` procedure, the value represented by the name, the value for the `dataSize` parameter, and the value for the `dataHandle` parameter. (Be sure to dispose of the memory you allocate for any handle you pass in the `dataHandle` parameter.) Keep in mind that it is impossible to determine which picture comments are supported by the driver of the current printer.

Table B-1 Names, values, and data sizes for picture comments

Name	Value	Data size	Data handle	Description
Text picture comments				
`TextBegin`	150	6	`TTxtPicRec`	Begin text function
`TextEnd`	151	0	`NIL`	End text function
`StringBegin`	152	0	`NIL`	Begin string delimitation
`StringEnd`	153	0	`NIL`	End string delimitation
`TextCenter`	154	8	`TCenterRec`	Offset to center of rotation for text
`LineLayoutOff`	155	0	`NIL`	Turn printer driver's line layout off
`LineLayoutOn`	156	0	`NIL`	Turn printer driver's line layout on
`ClientLineLayout`	157	16	`TClientLLRec`	Customize line layout error distribution

continued

Table B-1 Names, values, and data sizes for picture comments (continued)

Name	Value	Data size	Data handle	Description
Graphics picture comments				
`PolyBegin`	160	0	`NIL`	Begin special polygon
`PolyEnd`	161	0	`NIL`	End special polygon
`PolyIgnore`	163	0	`NIL`	Ignore following polygon data
`PolySmooth`	164	1	`TPolyVerbRec`	Close, fill, frame
`PolyClose`	165	0	`NIL`	Smooth the curve between endpoints
`RotateBegin`	200	8	`TRotationRec`	Begin rotated port
`RotateEnd`	201	0	`NIL`	End rotation
`RotateCenter`	202	8	`TCenterRec`	Offset to center of rotation
Line-drawing picture comments				
`DashedLine`	180	Size of a `TDashedLineRec` record	`TDashedLineRec`	Draw following lines as dashed
`DashedStop`	181	0	`NIL`	End dashed lines
`SetLineWidth`	182	4	`TLineWidthHdl`	Set fractional line widths
PostScript picture comments				
`PostScriptBegin`	190	0	`NIL`	Set driver state to PostScript
`PostScriptEnd`	191	0	`NIL`	Restore QuickDraw state
`PostScriptHandle`	192	Length of PostScript data	`Handle`	PostScript data referenced in handle
`PostScriptFile`	193	Length of PostScript data	`Handle`	Filename referenced in handle
`TextIsPostScript`	194	0	`NIL`	QuickDraw text is sent as PostScript
`ResourcePS`	195	8	Resource type, resource ID, index	PostScript data in a resource file
`PSBeginNoSave`	196	0	`NIL`	Set driver state to PostScript

Table B-1 Names, values, and data sizes for picture comments (continued)

Name	Value	Data size	Data handle	Description
Forms-printing picture comments				
`FormsPrinting`	210	0	`NIL`	Don't clear print buffer after each page
`EndFormsPrinting`	211	0	`NIL`	End forms printing after `PrClosePage`
ColorSync picture comments				
`CMBeginProfile`	220	0	`NIL`	Begin ColorSync profile
`CMEndProfile`	221	0	`NIL`	End ColorSync profile
`CMEnableMatching`	222	0	`NIL`	Begin ColorSync color matching
`CMDisableMatching`	223	0	`NIL`	End ColorSync color matching

All PostScript LaserWriter drivers support the picture comments listed in Table B-1.

Some third-party QuickDraw printer drivers support the `TextBegin`, `TextCenter`, and `TextEnd` picture comments.

The QuickDraw LaserWriter SC driver supports the `LineLayoutOff`, `LineLayoutOn`, and `SetLineWidth` picture comments.

The QuickDraw ImageWriter LQ driver and versions prior to 7.2 of the QuickDraw StyleWriter driver support the `LineLayoutOff` and `LineLayoutOn` picture comments.

The QuickDraw Personal LaserWriter LS driver and versions later than 7.2 of the QuickDraw StyleWriter driver support no picture comments at all.

The `SetGrayLevel` picture comment is now obsolete. The `PostScriptFile`, `TextIsPostScript`, `FormsPrinting`, `EndFormsPrinting`, `ClientLineLayout`, `PSBeginNoSave`, and `ResourcePS` picture comments have limited use and are no longer recommended.

See *Inside Macintosh: Advanced Color Imaging* for information about the picture comments used by the ColorSync Utilities.

Maintaining Device Independence

Whenever printing, you should use both QuickDraw and non-QuickDraw representations of an image, so that the current printer driver can render the best possible picture. If you send an image described with picture comments to a QuickDraw printer driver that does not support those picture comments, the driver ignores the comments and subsequently does not print your image; if you send only a QuickDraw image to a printer driver that supports picture comments, the driver may not render its best possible image.

Printer drivers that support `TextBegin`, `TextCenter`, and `TextEnd` are expected to ignore calls to the `CopyBits`, `CopyMask`, and `CopyDeepMask` procedures that fall between the `TextBegin` and `TextEnd` picture comments. Between the `TextBegin` and `TextEnd` picture comments, you can use `CopyBits` to draw a bitmap representation of rotated text on QuickDraw printers; this bitmap is not used if the `TextBegin` and `TextEnd` picture comments are supported, but it is used if `TextBegin` and `TextEnd` are not supported. This is illustrated in Listing B-4 on page B-21.

When your application draws polygons on a PostScript printer, you can use `PolyBegin`, `PolySmooth`, and `PolyEnd` picture comments to draw smoothed polygons; QuickDraw printer drivers ignore these comments. To make a PostScript printer driver ignore your QuickDraw representation of the polygons, you can use the `PolyIgnore` picture comment, as illustrated in Listing B-6 on page B-27.

A technique for maintaining two sets of drawing codes, described in "Rotating Graphics" beginning on page B-29 and "Drawing Dashed Lines" beginning on page B-33, makes use of a "magic pen" visible only to PostScript drivers. Graphics comments for drawing dashed lines and for rotating graphics require the use of the `PenMode` procedure to set the pattern mode to a value of 23. Normally this value is undefined, but it is handled specially by PostScript printer drivers (all QuickDraw drivers ignore it). Your application can use this pattern mode to draw objects in a picture, and if the picture is printed on a QuickDraw printer, these objects are not visible.

To maintain device independence when you send routines to a PostScript printer driver, you can "hide" QuickDraw routines between the `PostScriptBegin` and `PostScriptEnd` picture comments. The `PostScriptBegin` comment is recognized only by PostScript printer drivers. When a PostScript driver receives the `PostScriptBegin` comment, it tells the PostScript printer to save the current state of the printer and to disable all low-level standard QuickDraw drawing procedures. Thus, the QuickDraw representation of the graphic is ignored by PostScript printer drivers.

Table B-2 lists the QuickDraw low-level procedures and the affected high-level drawing routines that are disabled by the `PostScriptBegin` picture comment.

Table B-2 Low-level QuickDraw routines disabled by the `PostScriptBegin` comment

Low-level routine	Examples of affected high-level QuickDraw routines
`StdText`	QuickDraw text-drawing routines (as described in the chapter "QuickDraw Text" in *Inside Macintosh: Text*)
`StdLine`	`MoveTo`, `Move`, `LineTo`, `Line`
`StdRect`	`FrameRect`, `PaintRect`, `FillRect`, `EraseRect`, `InvertRect`
`StdRRect`	`FrameRoundRect`, `PaintRoundRect`, `FillRoundRect`, `EraseRoundRect`, `InvertRoundRect`
`StdOval`	`FrameOval`, `PaintOval`, `FillOval`, `EraseOval`, `InvertOval`
`StdArc`	`FrameArc`, `PaintArc`, `FillArc`, `EraseArc`, `InvertArc`
`StdPoly`	`FramePoly`, `PaintPoly`, `FillPoly`, `ErasePoly`, `InvertPoly`
`StdRgn`	`FrameRgn`, `PaintRgn`
`StdBits`	`CopyBits`, `CopyMask`, `CopyDeepMask`

To mark the end of a sequence of hidden QuickDraw drawing routines and to reenable QuickDraw drawing routines, you can use the picture comment `PostScriptEnd`. The `PostScriptEnd` comment is recognized only by PostScript printer drivers. When a PostScript driver receives the `PostScriptEnd` comment, it tells the PostScript printer driver to restore the previous state of the printer driver and to enable QuickDraw drawing operations.

For a LaserWriter PostScript printer driver, QuickDraw routines that draw text, lines, and shapes and copy bitmaps or pixel maps have no effect when placed between the `PostScriptBegin` and `PostScriptEnd` picture comments. Instead, the driver expects to receive imaging instructions in subsequent picture comments. On the other hand, a QuickDraw printer driver ignores the `PostScriptBegin` and `PostScriptEnd` picture comments.

Only PostScript printer drivers should support the `DashedLine`, `DashedStop`, `RotateBegin`, `RotateCenter`, and `RotateEnd` picture comments. Therefore, you can use the `PostScriptBegin` and `PostScriptEnd` picture comments to hide your QuickDraw implementations of these comments from the printer driver. Listing B-7 on page B-31 illustrates how to use `PostScriptBegin` and `PostScriptEnd` when rotating graphics on PostScript printers; Listing B-9 on page B-34 illustrates how to use `PostScriptBegin` and `PostScriptEnd` when drawing dashed lines on PostScript printers.

Synchronizing QuickDraw and PostScript Printer Drivers

QuickDraw instructions such as those generated by the `Move`, `MoveTo`, `PenPat`, and `PenSize` routines change the state of the current graphics port without going through the standard low-level routines pointed to in the `QDProcs` record for the current graphics port. A printer driver takes these changes into account only at the time it executes an actual drawing instruction. The printer driver uses the routines specified in the `QDProcs` record at execution time and responds only to those instructions handled by the routines in the `QDProcs` record. Therefore, you should flush the state of the printing graphics port explicitly by calling any routine that goes through the `QDProcs.lineProc` field, as shown in Listing B-1, before inserting code using picture comments for a PostScript driver. The use of the application-defined routine `MyFlushGrafPortState` shown here is further illustrated in Listing B-8 on page B-32.

Listing B-1 Synchronizing QuickDraw and the PostScript driver

```
PROCEDURE MyFlushGrafPortState;
VAR
   penInfo: PenState;
BEGIN
   GetPenState(penInfo);   {save pen size}
   PenSize(0,0);           {make it invisible}
   MoveTo(-3200,-3200);    {move the pen way off the page in }
                           { case the printer driver draws a dot }
                           { even with a pen size of (0,0)}
   Line(0,0);              {go through QDProcs.lineProc}
                           {next, restore pen size}
   PenSize(penInfo.pnSize.h, penInfo.pnSize.v);
END;
```

A PostScript printer driver separates the PostScript code generated for text-drawing instructions (which usually involves font queries and, sometimes, font downloading) from the picture comments intended for PostScript devices. In certain cases, this results in apparently nonsequential execution of drawing instructions and may affect clipping regions or have side effects on the drawing operations you include in picture comments. To synchronize the sequence of QuickDraw routines with the generation of PostScript code, you need to flush the buffer maintained by the PostScript driver. You can do this by using the `PostScriptBegin` picture comment followed immediately by the `PostScriptEnd` picture comment. This causes all PostScript code, generated either by the application or by the printer driver, to be sent to the printer. Listing B-2 shows an application-defined procedure that does this. The use of the application-defined routine `MyFlushPostScriptState` shown here is further illustrated in Listing B-4 on page B-21.

Listing B-2 Flushing the buffer for a PostScript printer driver

```
PROCEDURE MyFlushPostScriptState;
BEGIN
   PicComment(PostScriptBegin, 0, NIL);
   PicComment(PostScriptEnd, 0, NIL);
END;
```

Using Text Picture Comments

The text picture comments listed in Table B-1 on page B-5 allow you to disable the printer driver's line layout capabilities (as described in the next section), construct lines of text out of disparate strings (as described in "Delimiting Strings" on page B-16), and rotate text on the page (as described in "Rotating Text" on page B-17).

For information on drawing text, see *Inside Macintosh: Text*.

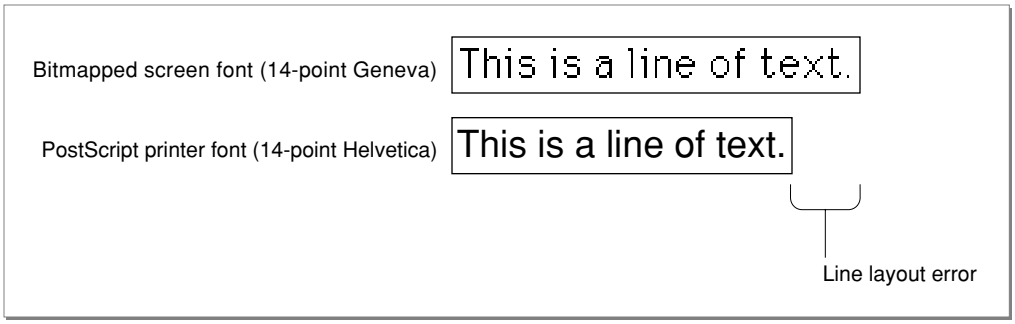
Disabling and Reenabling Line Layout

When your application draws text into a printing graphics port, the printer driver may do a lot of extra work depending on the current printer; the printer driver may have to scale and smooth fonts, remap characters, and substitute one font used onscreen for another that exists on the printer (this last action is called **font substitution**).

After it selects the appropriate font, the printer driver matches the width of the printed line with the width of the screen line. If the driver has to perform font substitution, the two lines may be very different. For example, if your application draws a document with the Geneva bitmapped font (instead of the Geneva TrueType font), a PostScript printer

driver could substitute the Helvetica® font for Geneva in the PostScript code it generates. Since Helvetica is a different font, it has different metrics. A rather exaggerated example of the effects of font substitution can be found in Figure B-1.

Figure B-1 The line layout error between a bitmapped font and a PostScript font



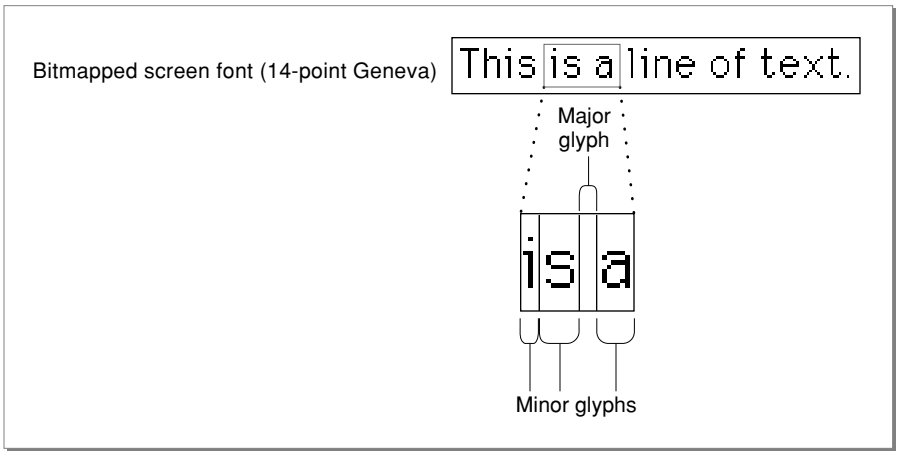
For the typical user, the appearance of Helvetica on the printed page is not that much different from the appearance of Geneva on the screen. However, the width of the lines using the two fonts is different; this difference is called the **line layout error.** The line of text using the bitmapped screen font is much wider than the line of text using the PostScript printer font. (Depending on the font used in the document or substituted on the printer, you might also run into cases where the screen width is narrower than the printed width.)

Note
There are no line layout problems with TrueType fonts, unless one font has the same name as—but a different character width from—a printer-resident PostScript font. ◆

To distribute the layout error, a printer driver must effectively increase or decrease the width of each glyph in the line. A **glyph** is the distinct representation of a character in a form that a screen or printer can display. A glyph may represent one character (the lowercase *a*), more than one character (the *fi* ligature, two characters but one glyph), or a nonprinting character (the space character). When using Roman scripts, most lines of text contain some number of space character glyphs. Printer drivers take advantage of this fact and normally apply most of the layout error to space glyphs (known as the **major glyphs**) and the rest of the error to the other glyphs in the string (known as the **minor glyphs**).

In Figure B-2, the *i*, *s*, and *a* characters are examples of minor glyphs, where *s* and *a* are separated by the major glyph (the space character).

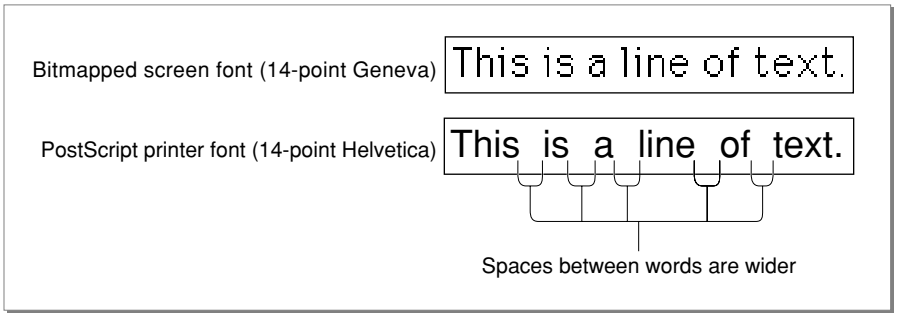
Figure B-2 Major and minor glyphs



The amount of error applied to the major glyph is known as the **major error,** and the amount applied to the other glyphs is the **minor error.**

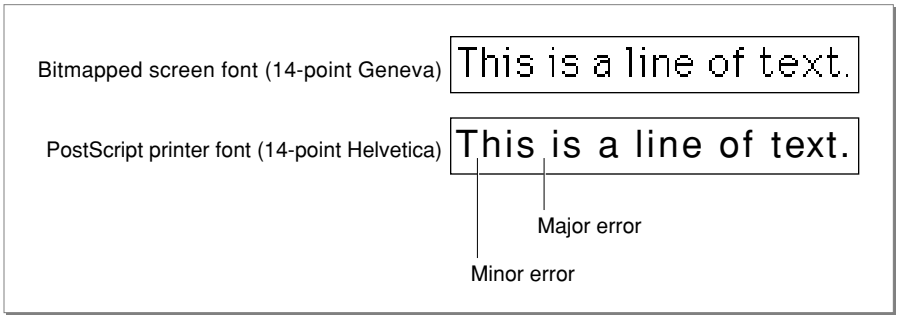
In Figure B-3, the printer driver corrects most of the difference between the line widths by expanding the width of the space glyphs in the string.

Figure B-3 Distributing layout error to the major glyphs



However, if the printer driver expands only the width of the spaces, the line has a strange appearance. To balance the changes made to the space glyphs, the driver's line layout routines increase the space between each glyph in the string by a small amount. After the line is laid out in this way, the printed string should be almost exactly as wide as the string that was displayed on the screen. As shown in Figure B-4, the space between the uppercase *T* and the lowercase *h* in the word *This* has been increased, but only slightly; most of the error has been applied to the spaces. By default, most drivers apply about 80 percent of the total line layout error to the major glyphs and the other 20 percent to the minor glyphs. When using a script system that does not use the space glyph to delimit words, the layout error is distributed evenly across all characters in the line.

Figure B-4 Distributing layout error among major and minor glyphs



A printer driver's line layout routines are device-dependent. Since different devices have different resident fonts, the layout error can be quite large. For this reason, you should not assume that if you have the correct output on one type of laser printer you will have the correct output on all devices or with all fonts.

Although the printer driver can compute the placement of a line of text on the page so that it closely approximates the placement of the line on the screen, there are times when adjusting the line of text by adding space can have an adverse effect on the line layout that your application has already done.

You can disable the line layout routines of the current printer driver and give your application more control over placement of the glyphs on the page by using the `LineLayoutOff` picture comment. You may want to use this picture comment if your application prints monospaced, tab-formatted text; draws notes or other music symbols using glyphs from a music font; or renders mathematical equations or formulas. For example, if your application displays musical notation, the notes should stay where your application placed them, because small shifts in position can cause the music to be misread.

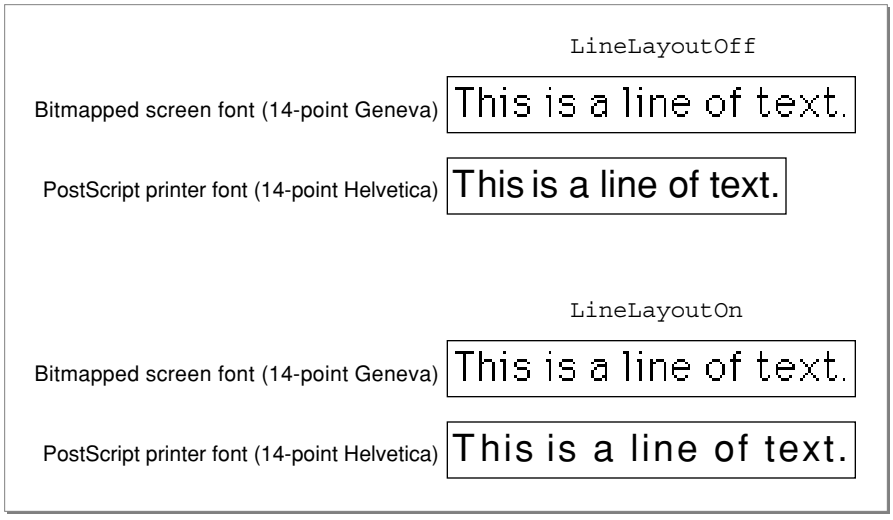
The `LineLayoutOff` picture comment instructs the printer driver to make no adjustments to the text being sent. Your application is then responsible for identically matching the appearance of text displayed on the screen to the printer. If the current printer driver does not support these comments, it ignores them and places the text on the page as well as it can.

You can reenable the printer driver's line layout routines with the `LineLayoutOn` picture comment (however, some printer drivers support only the `LineLayoutOff` comment). Although general line layout is disabled, some small shifts in glyph position may still occur. These shifts are usually not a problem, but, if they are, you should use the `PrGeneral` procedure with the `getRslDataOp` and `setRslOp` opcodes (described in the chapter "Printing Manager" in this book) to draw text at the resolution of the current printer.

IMPORTANT
Setting the `FractEnable` global variable (described in the chapter "Font Manager" in *Inside Macintosh: Text*) to `TRUE` does not have precisely the same effect as using the `LineLayoutOff` picture comment. You should explicitly use the `LineLayoutOff` picture comment rather than the `SetFractEnable` procedure. ▲

Figure B-5 compares the results of an application using the `LineLayoutOff` picture comment and the `LineLayoutOn` picture comment. In the first example, the text is printed exactly as it is rendered on the printer, with a much smaller width. In the second example, the printer driver's line layout routines make the screen and printer lines the same length.

Figure B-5 Using the `LineLayoutOff` and `LineLayoutOn` picture comments



In computing the required line layout adjustments, the PostScript LaserWriter driver proceeds as follows:

1. It collects text processed by the routine pointed to in the `textProc` field of the printing graphics port's `QDProcs` record, and assembles the text into a logically contiguous line. This includes text moved vertically away from the baseline to take care of diacritical marks or exponents in the text. The accumulation of text stops when the PostScript LaserWriter driver detects that the pen position has moved horizontally since the conclusion of the previous text-drawing instruction, or when the driver encounters picture comments such as `TextBegin`, `TextEnd`, `StringBegin`, and `StringEnd`.
2. It determines the width of the accumulated logical line of text, both on the screen and on the printer, and distributes the line layout error among the interword and intercharacter spacing of the printed output.

The `LineLayoutOff` picture comment disables only the second step (distribution of the line layout error); the algorithm of accumulating text into a logically contiguous piece is not affected. Otherwise, if the character widths of the printer font are different from those of the screen font, and if the text contains diacritical marks or exponents, the diacritical marks and exponents would often be misplaced.

If you want precise control over the placement of different text strings within a line, you must override the heuristic line accumulation algorithm of the PostScript LaserWriter driver (described in the first step). A good way to override this algorithm is to use the `StringBegin` and `StringEnd` picture comments to mark individual strings as logically independent text entities; this prevents the PostScript LaserWriter driver from assembling the strings into one logically contiguous line of text. The `StringBegin` and `StringEnd` picture comments are described in the next section; Listing B-3 on page B-17 illustrates how to completely disable line layout by using the `LineLayoutOff` and `StringBegin` picture comments.

Delimiting Strings

You may want to draw a particular text string in pieces instead of a whole. For example, to draw kerned glyphs, you can draw the first part of the string—up to the point where kerning occurs—using the `DrawText` procedure, and you can then adjust the pen and draw the kerned glyph using the `DrawChar` procedure. (The `DrawText` and `DrawChar` procedures are described in the chapter "QuickDraw Text" in *Inside Macintosh: Text*.) You can also draw a single string that contains different fonts, styles, or sizes—if you call `DrawText` each time the typeface or font style changes. To identify the beginning of a single string that will be drawn using multiple calls to a QuickDraw text-drawing routine, you can use the `StringBegin` picture comment. Use the `StringEnd` picture comment to mark its end.

You can use the `StringBegin` and `StringEnd` picture comments if your application needs complete control over glyph placement on a page. If your application uses text-editing boxes for individual strings, it can use these picture comments to treat each string as a separate piece of text and place all glyphs into one text-editing box.

Listing B-3 uses the `StringBegin` and `StringEnd` picture comments. Use the `LineLayoutOff` picture comment (described in the preceding section) in conjunction with the `StringBegin` comment to turn line layout completely off.

Listing B-3 Disabling line layout by using the `LineLayoutOff` and `StringBegin` picture comments

```
PROCEDURE MyStringReconDemo (x: XArray; y: Integer);
BEGIN
   PicComment(LineLayoutOff,0,NIL);
   PicComment(StringBegin,0,NIL);
   {position each character of the word 'Test' using }
   { MoveTo and DrawChar}
   MoveTo(x[1],y); DrawChar('T');
   MoveTo(x[2],y); DrawChar('e');
   MoveTo(x[3],y); DrawChar('s');
   MoveTo(x[4],y); DrawChar('t');
   {reenable the printer driver's line layout routines}
   PicComment(StringEnd,0,NIL);
   PicComment(LineLayoutOn,0,NIL);
END;
```

Rotating Text

You can use picture comments to rotate text on PostScript devices and on any QuickDraw-based drivers that support text rotation. (This is not the kind of rotation associated with landscape and portrait orientation of the printer paper as selected by the user through the style dialog box. This rotation occurs in reference to the current QuickDraw graphics port only.) The picture comments to rotate text are `TextBegin`, `TextCenter`, and `TextEnd`.

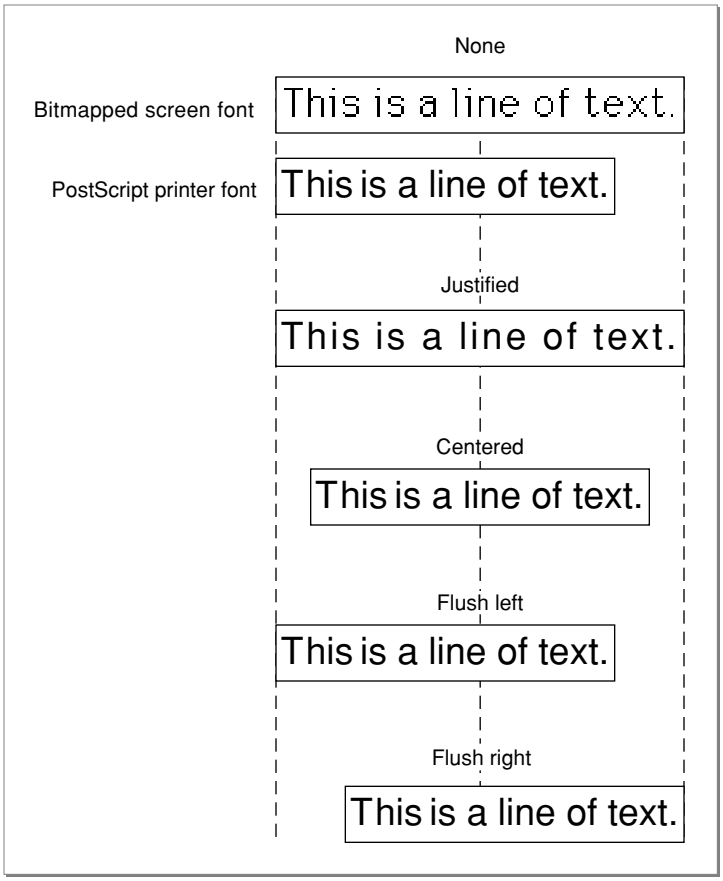
If you use picture comments to rotate text, you should also generate a device-independent representation, such as a bitmapped version of the text, to be used on QuickDraw devices that don't support these picture comments. Printer drivers that support `TextBegin`, `TextCenter`, and `TextEnd` are expected to ignore calls to the `CopyBits`, `CopyMask`, and `CopyDeepMask` procedures (as well as QuickDraw clipping regions) between the `TextBegin` and `TextEnd` picture comments. In this way, you can use `CopyBits` to draw a bitmap representation of rotated text on QuickDraw printers; the bitmap is not used if the `TextBegin` and `TextEnd` picture comments are supported, but it is used if `TextBegin` and `TextEnd` are not supported.

Some versions of 2-byte Kanji systems print Kanji glyphs by calling the `CopyBits` procedure instead of calling standard text-drawing routines. You cannot use the text rotation picture comments with these fonts. Instead, use the picture comments described in "Rotating Graphics" beginning on page B-29.

To use picture comments to rotate text, you begin by specifying the amount of rotation as a parameter to the `TextBegin` comment. Next, you pass the center of rotation in the `TextCenter` comment. The printer driver rotates any text drawn between the `TextCenter` and `TextEnd` comments.

The `TextBegin` picture comment allows your application to specify left, right, center, or full justification; horizontal or vertical flipping; and degrees of rotation. The possible types of alignment are shown in Figure B-6.

Figure B-6 Variations in text alignment



When you specify the `TextBegin` picture comment in the `kind` parameter of the `PicComment` procedure, you also specify a `TTxtPicHdl` handle (a handle to a `TTxtPicRec` record) in the `dataHandle` parameter. Here is how you should declare these as Pascal data types in your application:

```
TYPE
   TTxtPicHdl  = ^TTxtPicPtr;
   TTxtPicPtr  = ^TTxtPicRec;
   TTxtPicRec  =
   PACKED RECORD
      tJus:          Byte;    {justification of text}
      tFlip:         Byte;    {horizontal or vertical flipping}
      tAngle:        Integer; {0..360 degrees clockwise rotation }
                              { in integer format}
      tLine:         Byte;    {reserved}
      tCmnt:         Byte;    {reserved}
      tAngleFixed:   Fixed;   {0..360 degrees clockwise rotation }
                              { in fixed-number format}
   END;
```

You supply the `tJus` field with one of these constants to specify the alignment setting of the text:

```
   CONST
      tJusNone    = 0;  {no alignment}
      tJusLeft    = 1;  {flush left}
      tJusCenter  = 2;  {centered}
      tJusRight   = 3;  {flush right}
      tJusFull    = 4;  {full justification}
```

Setting the `tJus` field to left, right, or centered tells the printer driver to maintain only the left, right, or center point of the line (respectively), preventing the driver from recalculating the interword spacing. A value of `tJusFull` specifies that both endpoints of the line must be maintained, so the driver recalculates interword spacing instead of rejustifying text.

You supply the `tFlip` field with one of these constants to specify the horizontal or vertical flipping of text about the center point (which, in turn, is specified with the `TextCenter` picture comment):

```
   CONST
      tFlipNone         = 0; {no flip of text}
      tFlipHorizontal   = 1; {horizontal flip of text}
      tFlipVertical     = 2; {vertical flip of text}
```

You supply the `tAngle` field with an integer to specify the number of degrees by which the printer driver should rotate the text.

The `tLine` and `tCmnt` fields are reserved.

You supply the `tAngleFixed` field with a fixed-point number to specify the number of degrees by which the printer driver should rotate the text.

In a `TTxtPicRec` record, you can provide the degrees of rotation both as an integer (in the `tAngle` field) and as a fixed-point number (in the `tAngleFixed` field). You should always specify the rotation in both fields, even for drivers that support only integral rotation. The driver determines which field to use based on the size of the handle passed to `PicComment`. If you do not define the `tAngleFixed` field in the `TTxtPicRec` record, the printer driver automatically uses the `tAngle` field.

To rotate an object, a printer driver needs information concerning the center of rotation. Immediately after a `TextBegin` comment, the driver expects the `TextCenter` picture comment specifying the offset to the center of rotation for any text enclosed within the text picture comments. The driver stores this offset and adds it to the location of the first text-drawing routine after it receives the `TextCenter` picture comment. This allows you to send multiple runs of text to be rotated with different centers of rotation, while using only one set of `TextBegin` and `TextEnd` picture comments. The printer driver expects the string locations to be in the coordinate system of the current graphics port.

The printer driver rotates the entire graphics port to draw the text, so it can draw several strings with one `TextBegin` picture comment and one `TextCenter` picture comment. You should always include as much text as possible in a single `TextBegin` picture comment so that the driver makes the fewest number of rotations.

The printer driver can draw nontextual objects within the bounds of the text rotation comments, but it must restore the printing graphics port to its original state to draw the object, and then rotate the printing graphics port again to draw the next string of text. You must send another `TextCenter` comment before each new rotation.

When you specify the `TextCenter` (or `RotateCenter`) picture comment in the `kind` parameter of the `PicComment` procedure, you also supply in the `dataHandle` parameter a `TCenterHdl` handle, which is a handle to a `TCenterRec` record. You can use this record to specify the center of rotation for text or (as described in "Rotating Graphics" beginning on page B-29) for graphics. Here is how you should declare these as Pascal data types in your application:

```
TYPE
   TCenterHdl  = ^TCenterPtr;
   TCenterPtr  = ^TCenterRec;
   TCenterRec  =
   RECORD
      y:    Fixed;   {vertical offset from current pen location}
      x:    Fixed;   {horizontal offset from current pen location}
   END;
```

You use the `y` field to specify the vertical offset along the y-axis from the current pen location to the center of rotation.

You use the `x` field to specify the horizontal offset along the x-axis from the current pen location to the center of rotation.

The application-defined routine `MyDrawXString`, shown in Listing B-4, rotates the strings by the degrees specified in the `rot` parameter. The rotation occurs around the current point, offset by the value passed in the `ctr` parameter. The strings are justified and flipped according to the `just` and `flip` parameters. If the printer driver supports the `TextBegin`, `TextCenter`, and `TextEnd` picture comments, the printer driver rotates the text at device resolution; otherwise, an application-defined procedure is called to generate a bitmap of the rotated and flipped text, using `CopyBits` to draw the text in the printing graphics port. The pen position is preserved. (Listing B-8 on page B-32 illustrates how to use the `TCenterRec` record to rotate graphics.)

Listing B-4 Displaying rotated text using picture comments

```
PROCEDURE MyDrawXString(s: Str255; ctr: Point;
                        just, flip: Integer; rot: Fixed);
VAR
   hT:          TTxtPicHdl;
   hC:          TCenterHdl;
   zeroRect:    Rect;
   pt:          Point;
   oldClip:     RgnHandle;
BEGIN
   GetPen(pt);    {to preserve the pen position}
   hT := TTxtPicHdl(NewHandle(SizeOf(TTxtPicRec)));
   hC := TCenterHdl(NewHandle(SizeOf(TCenterRec)));
   WITH hT^^ DO
   BEGIN
      tJus := just;
      tFlip := flip;
      tAngle := - FixRound(rot); {counterclockwise}
      tLine := 0; {reserved}
      tCmnt := 0; {used internally by the printer driver}
      tAngleFixed := - rot;
   END;
   hC^^.y := Long2Fix(ctr.v);
   hC^^.x := Long2Fix(ctr.h);
   MyFlushPostScriptState; {see Listing B-2 on page B-11}
   PicComment(TextBegin,SizeOf(TTxtPicRec),Handle(hT));
   PicComment(TextCenter,SizeOf(TCenterRec),Handle(hC));
   {graphics state now has rotated/flipped coordinates}
```

```
   oldClip := NewRgn;
   GetClip(oldClip);
   SetRect(zeroRect,0,0,0,0);
   ClipRect(zeroRect);  {hides this DrawString from }
   DrawString(s);       { QuickDraw in the rotated }
                        { environment}
   ClipRect(oldClip^^.rgnBBox);
   {now the "fallback" bitmap representation}
   MyQDStringRotation(s, ctr, just, flip, rot);
   PicComment(TextEnd, 0, NIL);
   {set environment back to the original state}
   DisposeHandle(Handle(hT));
   DisposeHandle(Handle(hC));
   MoveTo(pt.h, pt.v);  {restore the pen position}
END;
```

Because the PostScript LaserWriter driver buffers generated PostScript code, and because the driver ignores clipping regions between the `TextBegin` and `TextEnd` picture comments, clipping regions for drawing instructions that precede `TextBegin` may be affected. Therefore, `MyDrawXString` uses the application-defined routine `MyFlushPostScriptState` (shown in Listing B-2 on page B-11) immediately before using the `TextBegin` picture comment.

Using Graphics Picture Comments

Graphics picture comments, listed in Table B-1 on page B-5, provide your application with the ability to render smoothed polygons (as described in the next section) and to rotate graphics (as described in "Rotating Graphics" on page B-29).

In general, you cannot use one set of graphics picture comments (for instance, the polygon-drawing picture comments) with another (graphics rotation comments). When using these two types of comments, you should simply rotate the points of the polygon before drawing.

The graphics comments for drawing dashed lines and for rotating graphics require the use of the `PenMode` procedure (described in the chapter "QuickDraw Drawing" in this book) to set the pattern mode to a value of 23. Normally this value is undefined, but it is handled specially by PostScript printer drivers, which treat it like the `srcCopy` Boolean transfer mode (described in the chapters "QuickDraw Drawing" and "Color QuickDraw"). All QuickDraw drivers ignore this pattern mode. Your application can use this pattern mode to draw objects in a picture and, if the picture is printed on a QuickDraw printer, these objects are not visible.

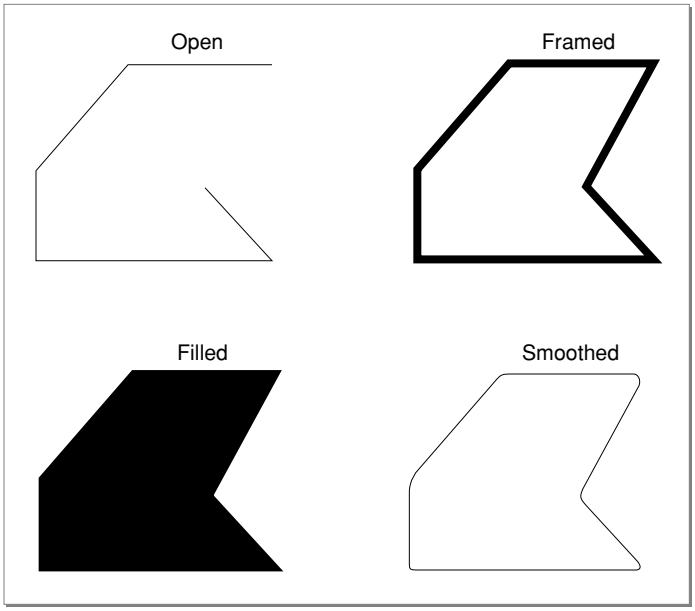
Drawing Polygons

By using picture comments, you can draw high-resolution polygons on PostScript printing devices. PostScript supports four types of polygons: open, framed, filled, and smoothed. (QuickDraw supports all of these types except smoothed.)

Type	Description
Open	A polygon whose endpoints do not join. This type of polygon cannot be filled.
Framed	A closed polygon that is not filled. Framed and filled polygons are exclusive to one another.
Filled	A closed polygon whose interior is entirely covered with a pattern.
Smoothed	A polygon (open, framed, or filled) whose edges have been rounded.

Figure B-7 shows these four types of polygons.

Figure B-7 Types of polygons



To draw polygons, perform the following steps:

1. Use the `PolyBegin` picture comment to alert the PostScript driver that you are drawing a polygon.
2. Optionally, you can use the `PolyClose` picture comment to use "closed" smoothing between the first and last vertices of the polygon.
3. Use the `PolySmooth` picture comment to tell the PostScript driver to draw a Bézier curve.
4. Use the `GetClip` procedure to save the current clipping region; then use the `ClipRect` procedure to hide your polygon's drawing commands from QuickDraw.
5. Draw your polygon. The PostScript driver renders it smoothly.
6. Use the `SetClip` procedure to restore the previous clipping region.
7. Use the `PolyIgnore` picture comment to make the printer driver ignore the line-drawing commands for your QuickDraw representation of the polygon.
8. Draw your QuickDraw representation of the polygon.
9. Use the `PolyEnd` picture comment.

The `PolyBegin` and `PolyEnd` picture comments surround the polygon description. Note that the printer driver draws the polygon at the location of the pen when it receives the `PolyBegin` picture comment, so you must set the pen's location before using the `PolyBegin` picture comment. For polygons that are smoothed, you must set the pen size to 0 after the `PolyBegin` picture comment to prevent the unsmoothed polygon from being drawn on printers that do not support the polygon comments.

All QuickDraw routines called between `PolyBegin` and `PolyEnd` that are processed by the low-level `StdLine` routine are part of the polygon—that is, the endpoints of each of the lines become vertices of the polygons.

You should use the `PolyClose`, `PolySmooth`, and `PolyIgnore` picture comments between the `PolyBegin` and `PolyEnd` picture comments.

The `PolyClose` comment specifies that the printer driver should treat all vertices of the polygon in the same manner; in particular, this affects the shape of the smooth curve between the polygon's first and last vertices, which might otherwise be distinguishable as separate points. The `PolyClose` comment, however, does not automatically close the polygon as the PostScript operator `closepath` does.

To render high-resolution B-splines when PostScript is available, use the `PolySmooth` picture comment, which directs the PostScript printer driver to interpret the polygon vertices as control nodes for a quadratic Bézier spline. PostScript has a direct facility for cubic B-splines, and the PostScript printer driver translates the quadratic B-spline nodes into the appropriate nodes for a cubic B-spline that will emulate the original quadratic. This allows you to use this PostScript feature without having to call PostScript routines directly.

Note
PostScript Level 1 has some problems with very large polygons that have more than 1500 points. For this reason, you may want to avoid doubling the points on large smoothed polygons, even though a greater number of points might aid in making the polygon smoother. ◆

When you use the `PolySmooth` picture comment, pass a `TPolyVerbHdl` handle, which is a handle to a `TPolyVerbRec` record, in the `dataHandle` parameter of the `PicComment` procedure. You use a `TPolyVerbRec` record to tell the printer driver to interpret the polygon vertices as control nodes for a quadratic Bézier spline. Here is how you should declare these as Pascal data structures in your application:

```
Type
   TPolyVerbHdl   = ^TPolyVerbPtr;
   TPolyVerbPtr   = ^TPolyVerbRec;
   TPolyVerbRec   =
   PACKED RECORD
      f7,f6,f5,f4,f3:   Boolean;    {reserved; set to 0}
      fPolyClose:       Boolean;    {TRUE is same as PolyClose }
                                    { picture comment}
      fPolyFill:     Boolean;       {TRUE means fill polygon}
      fPolyframe:    Boolean;       {TRUE means frame polygon}
   END;
```

The `f7`, `f6`, `f5`, `f4`, and `f3` fields are reserved bits; you should set them to 0.

Setting the `fPolyClose` field to 1 achieves the same result as the `PolyClose` picture comment. The `PolyClose` comment specifies that the printer driver should treat all vertices of the polygon in the same manner; in particular, this affects the shape of the smooth curve between the polygon's first and last vertices, which might otherwise be distinguishable as separate points. The `PolyClose` comment does not automatically close the polygon as the PostScript operator `closepath` does.

Set the `fPolyFill` field to 1 if you want the printer driver to fill the polygon, or set it to 0 if not.

Set the `fPolyFrame` field to 1 if you want the printer driver to frame the polygon, or set it to 0 if not.

In Listing B-5, the polygon coordinates are defined through arrays of points, initialized using an application-defined procedure, `MyDefineVertices`. The procedure `MyDefineVertices` specifies the points for two polygons. The array referenced through the parameter `p` defines the points used for the PostScript representation of the polygon. The array referenced through the parameter `q` defines the points used for the QuickDraw representation of the polygon.

Listing B-5 Creating polygons

```
PROCEDURE MyDefineVertices(VAR p,q: PointArrayPtr);
CONST
   cx = 280;   {x coordinate for center point}
   cy = 280;   {y coordinate for center point}
   r0 = 200;   {radius}
   kN = 4;     {number of vertices for PostScript}
   kM = 6;     {number of vertices for QuickDraw approximation}
BEGIN
   {the array p^ contains the control points for the Bézier curve}
   SetPt(p^[0],cx + r0,cy);
   SetPt(p^[1],cx,cy + r0);
   SetPt(p^[2],cx - r0,cy);
   SetPt(p^[3],cx,cy - r0);
   p^[4] := p^[0];
   {q^ contains the points for a QuickDraw approximation of the curve}
   q^[0] := p^[0];
   SetPt(q^[1],cx,cy + round(0.7 * (p^[1].v - cy)));
   SetPt(q^[2],(p^[1].h + p^[2].h) DIV 2,
         (p^[1].v + p^[2].v) DIV 2);
   SetPt(q^[3],cx + round(0.8 * (p^[2].h - cx)),cy);
   SetPt(q^[4],q^[2].h,cy + cy - q^[2].v);
   SetPt(q^[5],q^[1].h,cy + cy - q^[1].v);
   q^[6] := q^[0];
END;
```

Use the `PolyIgnore` comment before drawing your QuickDraw version of the polygon; between `PolyIgnore` and `PolyEnd`, drivers that support these two comments ignore all QuickDraw routines processed through the low-level procedure `StdLine`. You can enclose the application-defined procedure `MyPolygonDemo`, shown in Listing B-6, between `OpenPicture` and `ClosePicture` calls to create a picture containing both QuickDraw and PostScript representations of the polygon. Alternatively, you can call `MyPolygonDemo` when drawing directly into a printing graphics port.

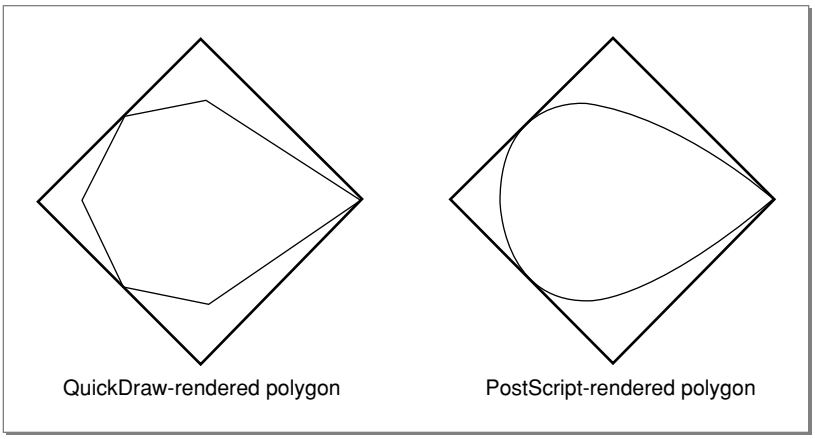
Listing B-6 Drawing polygons

```
PROCEDURE MyPolygonDemo;
VAR
   p, q:                PointArrayPtr;
   aPolyVerbH:          TPolyVerbHdl;
   i:                   Integer;
   clipRgn, polyRgn:    RgnHandle;
   zeroRect:            Rect;
BEGIN
   p := PointArrayPtr(NewPtr(SizeOf(Point) * (kN + 1)));
   q := PointArrayPtr(NewPtr(SizeOf(Point) * (kM + 1)));
   IF (p = NIL) OR (q = NIL) THEN DoErr(kMemError);
   MyDefineVertices(p,q);
   PenNormal;  {first show the standard QuickDraw polygon}
   MoveTo(p^[0].h,p^[0].v);
   FOR i := 1 TO kN DO
      LineTo(p^[i].h,p^[i].v);
   PenSize(2,2);  {now show the same polygon "smoothed"}
   PenPat(gray);
   {first, the PostScript representation, clipped from QuickDraw}
   aPolyVerbH:=
      TPolyVerbHdl(NewHandle(SizeOf(TPolyVerbRec)));
   IF aPolyVerbH<> NIL THEN
      WITH aPolyRecH^^ DO
      BEGIN
         fPolyFrame := TRUE;
         fPolyFill  := FALSE;
         fPolyClose := FALSE;
```

```
            {compare with the result for TRUE!}
            f3 := FALSE;
            f4 := FALSE;
            f5 := FALSE;
            f6 := FALSE;
            f7 := FALSE;
         END;
      MoveTo(p^[0].h,p^[0].v);
      PicComment(PolyBegin,0,NIL);
      {picComment(PolyClose,0,NIL); only if }
      { fPolyClose = TRUE, above!}
      PicComment(PolySmooth,SizeOf(TPolyVerbRec),
                 Handle(aPolyVerbH));
      clipRgn := NewRgn;
      GetClip(clipRgn);
      ClipRect(zeroRect);
      FOR i := 1 TO kN DO
         LineTo(p^[i].h,p^[i].v);
      {next, the QuickDraw approximation of the smoothed }
      { polygon, invisible for PostScript because of PolyIgnore}
      SetClip(clipRgn);
      PicComment(PolyIgnore,0,NIL);
      polyRgn := NewRgn;
      OpenRgn;
      MoveTo(q^[0].h,q^[0].v);
      FOR i := 1 TO kM DO
         LineTo(q^[i].h,q^[i].v);
      CloseRgn(polyRgn);
      FrameRgn(polyRgn);   {or FillRgn, if fPolyFill above is TRUE}
      PicComment(PolyEnd,0,NIL);
      DisposeHandle(Handle(aPolyVerbH));
      DisposeRgn(polyRgn);
      DisposePtr(Ptr(p));
      DisposePtr(Ptr(q));
END;
```

The two versions of the drawn polygon are shown in Figure B-8.

Figure B-8 QuickDraw and PostScript polygons



Note that you do not need to open a region, collect the line segments in the region, and draw the polygon through the `FrameRgn` procedure (described in the chapter "QuickDraw Drawing" in this book). This method is demonstrated in Listing B-6 only to prepare you for situations where you want to fill the polygon with a pattern. You cannot open a polygon and use the `FillPoly` procedure (also described in the chapter "QuickDraw Drawing" in this book), because the PostScript driver "owns" the polygon concept at this point and captures—and ignores—all line drawing between the `PolyIgnore` and `PolyEnd` comments. Regions do not interfere with polygons, however, and they can be used to paint or fill the polygonal shape.

Rotating Graphics

You can rotate QuickDraw objects on PostScript printers. The printer driver rotates the entire PostScript coordinate space before drawing the objects, which then appear rotated. All objects that you want to rotate must be contained between the `RotateBegin` and `RotateEnd` picture comments.

You specify the center of rotation with the `RotateCenter` picture comment. Unlike text rotation, where you pass the `TextBegin` picture comment first and then the `RotateCenter` picture comment, you must pass the offset (which is relative to the center of rotation) with the `RotateCenter` picture comment *before* you use the `RotateBegin` picture comment. When you specify the `RotateCenter` picture comment in the `kind` parameter of the `PicComment` procedure, you also supply in the `dataHandle` parameter a `TCenterHdl` handle, which is a handle to a `TCenterRec` record. You can use this record to specify the center of rotation for graphics or text. See "Rotating Text" beginning on page B-17 for a description of the fields of a `TCenterRec` record.

When you specify the `RotateBegin` picture comment in the `kind` parameter of the `PicComment` procedure, you also supply in the `dataHandle` parameter a `TRotationHdl` handle, which is a handle to a `TRotationRec` record. You use a `TRotationRec` record to specify the rotation of a graphic. Here's how you should declare these as Pascal data structures:

```
TYPE
   TRotationHdl = ^TRotationPtr;
   TRotationPtr = ^TRotationRec;
   TRotationRec =
   RECORD
      rFlip:         Integer; {horizontal/vertical flipping}
      rAngle:        Integer; {0..360 clockwise rotation in }
                              { integer format}
      rAngleFixed:   Fixed;   {0..360 clockwise rotation in }
                              { fixed-number format}
   END;
```

You use the `rFlip` field to specify whether to flip the graphic horizontally or vertically in addition to rotating it. Here are the possible values for this field:

Value	Description
0	No coordinate flip
1	Horizontal coordinate flip
2	Vertical coordinate flip

You supply the `rAngleFixed` field with a fixed-point number to specify the number of degrees by which the printer driver should rotate the graphic.

You can provide the degrees of rotation both as an integer (in the `rAngle` field) and as a fixed-point number (in the `rAngleFixed` field). You should always specify the rotation in both fields, even for drivers that support only integral rotation.

Once you set up the rotation with the `RotateCenter` and `RotateBegin` picture comments, you draw the graphics objects you want to rotate. Before drawing the objects, use the `PenMode` procedure to set the pattern mode to a value of 23, which represents a special pattern mode for PostScript printer drivers. You should draw the QuickDraw image, using the `CopyBits` procedure, inside its own pair of `PostScriptBegin` and `PostScriptEnd` comments so that the QuickDraw representation will not show up on PostScript devices. (You should also use the `PrGeneral` procedure with the `getRslDataOp` opcode, described in the chapter "Printing Manager" in this book, to determine and use the maximum printer resolution.)

In Listing B-7, the application-defined procedure `MyRotateDemo` rotates the same image for both QuickDraw and PostScript printers.

Listing B-7 Using picture comments to rotate graphics

```
PROCEDURE MyRotateDemo;
CONST
   angle = 30;
VAR
   spinRect:   Rect;
   delta:      Point;
BEGIN
   SetRect(spinRect,100,100,300,200);
   WITH spinRect DO SetPt(delta,(right - left) DIV 2,
                          (bottom - top) DIV 2);
   PenSize(2,2);
   PenPat(ltGray);
   FrameRect(spinRect); {show the unrotated square}
   PenNormal;
   MyPSRotatedRect(spinRect,delta,angle);
   {QuickDraw equivalent of the rotated object, hidden from the PostScript }
   { driver because of PostScriptBegin and PostScriptEnd}
   PicComment(PostScriptBegin, 0, NIL);
   MyQDRotatedRect(spinRect, delta, angle);
   PicComment(PostScriptEnd, 0, NIL);
END;
```

The application-defined procedure `MyQDRotatedRect` rotates the four points of the rectangle by an angle around the center and draws the rotated rectangle. To include this QuickDraw representation of the rotated objects (in case the `RotateCenter` and `RotateBegin` picture comments are not supported), the code in Listing B-7 assumes that only PostScript drivers implement these comments. The only way to hide from the driver the application-defined procedure that provides a QuickDraw representation of the rotated objects is to surround it by `PostScriptBegin` and `PostScriptEnd` comments.

To hide from QuickDraw the graphics rotation for a PostScript printer, Listing B-8 uses pattern mode 23.

Listing B-8 Using the `RotateCenter`, `RotateBegin`, and `RotateEnd` picture comments

```
PROCEDURE MyPSRotatedRect(r: Rect; offset: Point; angle: Integer);
{does the rectangle rotation for the PostScript LaserWriter driver}
{uses the RotateCenter, RotateBegin, and RotateEnd picture comments, }
{ and the "magic" pattern mode 23 to hide the drawing from QuickDraw}
CONST
   magicPen = 23;
VAR
   rInfo:      TRotationHdl;
   rCenter:    TCenterHdl;
   oldPenMode: Integer;
BEGIN
   rInfo := TRotationHdl(NewHandle(SizeOf(TRotationRec)));
   rCenter := TCenterHdl(NewHandle(SizeOf(TCenterRec)));
   IF (rInfo = NIL) OR (rCenter = NIL)
      THEN DebugStr('NewHandle failed');
   WITH rInfo^^ DO
   BEGIN
      rFlip := 0;
      rAngle := - angle;
      rAngleFixed := BitShift(LongInt(rAngle),16);
   END;
   WITH rCenter^^ DO
   BEGIN
      x := Long2Fix(offset.h);
      y := Long2Fix(offset.v);
   END;
   MoveTo(r.left,r.top);
   MyFlushGrafPortState;   {see Listing B-1 on page B-10}
   PicComment(RotateCenter,SizeOf(TCenterRec),Handle(rCenter));
   PicComment(RotateBegin,SizeOf(TRotationRec),Handle(rInfo));
   oldPenMode := thePort^.pnMode;
   PenMode(magicPen);
   FrameRect(r);
   PenMode(oldPenMode);
   PicComment(RotateEnd,0,NIL);
   DisposeHandle(Handle(rInfo));
   DisposeHandle(Handle(rCenter));
END;
```

Using Line-Drawing Picture Comments

Line-drawing picture comments, listed in Table B-1 on page B-5, provide your application with the ability to draw dashed lines (as described in the next section) and to display fractional line widths (as described in "Using Fractional Line Widths" on page B-35).

Drawing Dashed Lines

Your application may use dashed lines frequently, particularly if it is a spreadsheet or accounting application. You can use the `DashedLine` picture comment to draw dashed lines on capable printers without drawing each individual dash. You use the `DashedStop` picture comment to tell the printer driver when you are finished sending dashed line information.

When you use the `DashedLine` comment, the printer driver draws the indicated lines or rectangles. You should pass a handle to a `TDashedLineRec` record in the `dataHandle` parameter of the `PicComment` procedure. You use a `TDashedLineRec` record to specify how the dashed line should look. Here is how you should declare these as Pascal data structures:

```
TYPE
   TDashedLineHdl = ^TDashedLinePtr;
   TDashedLinePtr = ^TDashedLineRec;
   TDashedLineRec =
   PACKED RECORD
      offset:     SignedByte; {offset}
      centered:   SignedByte; {reserved; set to 0}
      intervals:  ARRAY[0..0] OF SignedByte;
                              {points for drawing and not }
                              { drawing dashes}
   END;
```

Use the `offset` field to specify an offset as with the PostScript `setdash` operator.

The `centered` field is reserved and should be set to 0. Your application must center the dashed lines.

In the `intervals` field, specify an array of dash intervals describing the number of points drawn for a dash and the number of points not drawn between them.

You must provide both a QuickDraw and a picture comment version of the dashed line. The code in Listing B-9 uses the `PostScriptBegin` and `PostScriptEnd` picture comments to hide QuickDraw code from PostScript, and it uses pattern mode 23 to render PostScript drawing invisible in QuickDraw.

Listing B-9 Using the `DashedLine` picture comment

```
PROCEDURE DashDemo;
CONST
   magicPen = 23;
   cx = 280;      {center along x-axis}
   cy = 280;      {center along y-axis}
   r0 = 200;      {radius}
VAR
   dashHdl:    TDashedLineHdl;
   i:          Integer;
   a, rad:     Extended;
BEGIN
   PenSize(2,2);
   {First the PostScript picture comment version. Pattern mode }
   { 23 makes the line drawing invisible to QuickDraw.}
   PenMode(magicPen);
   dashHdl := TDashedLineHdl(NewHandle(SizeOf(TDashedLineRec)));
   IF dashHdl <> NIL THEN
   WITH dashHdl^^ DO
   BEGIN
      offset := 4;       {just for fun}
      centered := 0;     {currently ignored--set to 0}
      intervals[0] := 2; {number of interval specs}
      intervals[1] := 4; {this means 4 points on ...}
      intervals[2] := 6; {... and 6 points off}
      PicComment(DashedLine, SizeOf(TDashedLineRec),
      Handle(dashHdl));
   END;
   rad := 3.14159 / 180;   {conversion degrees -> radians}
   FOR i := 0 TO 9 DO
   BEGIN    {draw some dashed lines}
      a := i * 20 * rad;
      MoveTo(cx, cy);
      Line(round(r0 * cos(a)), - round(r0 * sin(a)));
   END;
   PicComment(DashedStop, 0, NIL);  {that's enough!}
   DisposeHandle(Handle(dashHdl));
   PenMode(srcOr);  {no magic any more}
   {Now, the QuickDraw version. The PostScript driver must }
   { ignore it, so enclose it between PostScriptBegin and }
   { PostScriptEnd comments.}
   PicComment(PostScriptBegin, 0, NIL);
   PenSize(2,2);
```

```
   FOR i := 0 TO 9 DO
   BEGIN
      MoveTo(cx,cy);
      MyDashedQDLine(round(r0 * cos(i * 20 * rad)),
                     - round(r0 * sin(i * 20 * rad)), dashHdl);
   END;
   PicComment(PostScriptEnd, 0, NIL);
END;
```

Using Fractional Line Widths

Your application may need lines as thin as possible or thinner than the screen can display, especially if it is a desktop publishing, spreadsheet, or design application. You can draw **hairlines** (lines that are less than 1/72 of an inch wide) with printer drivers that support the `SetLineWidth` picture comment. Your application passes the printer driver a scaling factor (such as 1/4) that the driver applies to the pen size when rendering the picture.

QuickDraw and the PostScript language define 1 point to be 1/72 of an inch, so there are exactly 72 points per inch on the Macintosh screen. The resolution of a PostScript device such as the 300-dpi LaserWriter printer is about four times that of the screen, so the driver can render lines that are approximately 1/4 of a point thick, which is about 1/288 of an inch.

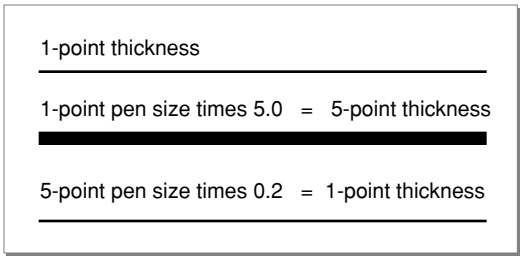
When you specify the `SetLineWidth` picture comment in the `kind` parameter of the `PicComment` procedure, you also specify a `TLineWidthHdl` handle (a handle to a data structure of type `TLineWidth`) in the `dataHandle` parameter. The `TLineWidth` data structure is defined by the `Point` data type. Here is how you should declare these as Pascal data types in your application:

```
TLineWidthHdl  = ^TLineWidthPtr;
TLineWidthPtr  = ^TLineWidth;
TLineWidth     = Point; {v = numerator, h = denominator}
```

Use the vertical coordinate of the point as the numerator and the horizontal coordinate as the denominator of the scaling factor: the driver multiplies the horizontal and vertical components of the pen by the scaling factor to obtain the new pen width. For example, if you have a pen size of (1,2) and your `SetLineWidth` picture comment uses 2 for the horizontal and 7 for the vertical, the pen size will then be $(7/2) \times 1$ pixel wide and $(7/2) \times 2$ pixels high.

In Figure B-9, the original pen size is 1 point. The first scaling factor is 5.0 or (5,1), which gives the pen a width of 5 points. The second scaling factor, applied to the new pen width, is 0.2 or (1,5), which gives the pen a width of 1 point again.

Figure B-9 Changing the pen width using the `SetLineWidth` picture comment



The `SetLineWidth` picture comment is implemented by all PostScript LaserWriter printer drivers and by some QuickDraw printer drivers. However, not all QuickDraw printer drivers support `SetLineWidth`, and there is no backup solution for cases where it is not supported. Among QuickDraw printer drivers that do support `SetLineWidth`, some drivers emulate PostScript printer drivers, while others—such as the QuickDraw LaserWriter SC driver—implement `SetLineWidth` differently.

The difference between the implementations of the `SetLineWidth` comment by the PostScript LaserWriter driver and the QuickDraw LaserWriter SC driver is apparent as soon as `SetLineWidth` is used a second time. The PostScript driver keeps an internal line-scaling factor, which is initialized to 1.0 when a job is started. Each number passed through `SetLineWidth` is multiplied by the current internal scaling factor to get the effective scaling factor for the pen size. The LaserWriter SC driver, on the other hand, replaces its current scaling factor for the pen size by the new value passed through `SetLineWidth`.

To support both implementations, you must always use an additional `SetLineWidth` picture comment to reset the PostScript driver line width to 1.0 before scaling to a new value width, as illustrated by the following lines of code:

```
PicComment(SetLineWidth, SizeOf(TLineWidth), Handle(1/oldLineWidth));
PicComment(SetLineWidth, SizeOf(TLineWidth), Handle(newLineWidth));
```

For example, suppose your application set the line width to 0.25, and now it needs a line width of 0.5. The following two `SetLineWidth` comments have the desired effect on all PostScript and QuickDraw drivers that implement the `SetLineWidth` comment.

Current line width, PS driver	Current line width, QD driver	Value passed along with `SetLineWidth`	New line width, PS driver	New line width, QD driver
0.25	0.25	4/1	1.0	4.0
1.0	4.0	1/2	0.5	0.5

The sample code in Listing B-10 gives the expected results on PostScript LaserWriter and QuickDraw printer drivers that implement the `SetLineWidth` comment.

Listing B-10 Using the `SetLineWidth` picture comment

```
PROCEDURE MySetNewLineWidth(oldWidth,newWidth: TLineWidth);
VAR
   tempWidthH: TLineWidthHdl;
BEGIN
   tempWidthH := TLineWidthHdl(NewHandle(SizeOf(TLineWidth)));
   tempWidthH^^.v := oldWidth.h;
   tempWidthH^^.h := oldWidth.v;
   PicComment(SetLineWidth, SizeOf(TLineWidth), Handle(tempWidthH));
   tempWidthH^^ := newWidth;
   PicComment(SetLineWidth, SizeOf(TLineWidth), Handle(tempWidthH));
   DisposeHandle(Handle(tempWidthH));
END;

PROCEDURE MyLineWidthDemo;
CONST
   y0 = 50;    {top left of demo}
   x0 = 50;
   d0 = 440;   {length of horizontal lines}
   e0 = 5;     {distance between lines}
   kN = 5;     {number of lines}
VAR
   oldWidth,newWidth:   TLineWidth;
   i,j,y:               Integer;
BEGIN
   PenNormal;
   y := y0;
   SetPt(oldWidth,1,1);             {initial line width = 1.0}
   FOR i := 1 TO 5 DO
   BEGIN
      SetPt(newWidth,4,i);
      {want to set it to i/4 = 0.25, 0.50, 0.75 ...}
      SetNewLineWidth(oldWidth,newWidth);
      MoveTo(x0, y);
      Line(d0, 0);
      y := y + e0;
      oldWidth := newWidth;
   END;
END;
```

Using PostScript Picture Comments

You can access the PostScript language directly using the `PostScriptHandle` picture comment, and so bypass QuickDraw entirely. When you send PostScript code directly to the printer driver, it sends your code directly to the printer with no preprocessing and no error checking.

Note
These picture comments affect the state of the PostScript drawing environment and can have such effects as printing blank pages. Also, many PostScript printer drivers do not use the same version of PostScript and produce different outputs with the same commands; you should test your code on as many PostScript printers as possible. In all cases, use the PostScript picture comments with extreme caution. ◆

Calling PostScript Routines Directly

Your application can tell the printer driver to disable all QuickDraw drawing routines by using the `PostScriptBegin` picture comment. The driver uses the PostScript `save` and `restore` operators to preserve the state of the PostScript interpreter. When the driver receives the `PostScriptEnd` picture comment, it reenables QuickDraw drawing routines.

You send PostScript code to the driver via the `PostScriptHandle` picture comment by including a handle to the PostScript code in the `dataHandle` parameter of the `PicComment` procedure. The driver performs no preprocessing or error checking on this code. The handle contains text with no length byte or word; use the `dataSize` parameter to convey the length of the PostScript code. (As with all picture comments, the handle you pass belongs to you, and you must dispose of it when you're finished with it.) You indicate the end of the PostScript commands with a carriage return (ASCII $0D). You must use `PostScriptBegin` and `PostScriptEnd` around any `PostScriptHandle` comments; otherwise, the PostScript driver will not properly save and restore the PostScript drawing environment.

Listing B-11 gives an example of an application-defined procedure called `DoPostScriptLine`. The procedure is used to transmit a string of PostScript code through the `PostScriptHandle` picture comment to the PostScript printer driver. `DoPostScriptLine` should be called only between `PostScriptBegin` and `PostScriptEnd` picture comments, as shown in the application-defined procedure `DoPostScriptComments`.

Listing B-11 Sending PostScript code directly to the printer

```
PROCEDURE DoPostScriptLine(s: Str255);
VAR
   h: Handle;
BEGIN
   h := NewHandle(256);
   IF h = NIL THEN DebugStr('NewHandle failed');
   BlockMove(@s[1], h^, Length(s));
   PicComment(PostScriptHandle, Length(s), h);
   h^^ := 13;
   PicComment(PostScriptHandle, 1, h); {add a carriage return}
   DisposeHandle(h);
END;

PROCEDURE DoPostScriptComments;
BEGIN
   {first, the simple example}
   PicComment(PostScriptBegin,0,NIL);
   DoPostScriptLine('100 100 moveto 0 100 rlineto 100 0 rlineto ');
   DoPostScriptLine('0 -100 rlineto -100 0 rlineto');
   DoPostScriptLine('stroke');
   MoveTo(30,30);
   DrawString('This text does not appear on PostScript printers.');
   PicComment(PostScriptEnd,0,NIL);
END;
```

Optimizing PostScript Printing

Although your printing code should be device-independent, you can optimize it for a PostScript printer. However, you cannot be sure that the current printer is a PostScript printer, so you may need to create two versions of the same drawing code: one for a PostScript printer and one for a QuickDraw printer, as described previously in this appendix.

For printing to a PostScript printer, you'll need to observe the following limitations:

- Regions aren't supported; try to simulate them with polygons or bitmaps.
- Clipping regions should be limited to rectangles. PostScript clips nonsquare patterns to squares.
- The `Invert` data type, part of the QuickDraw `GrafVerb` data type, is not supported by the PostScript LaserWriter printer driver.
- The PostScript LaserWriter driver does not support all Boolean transfer modes. It supports the `srcCopy`, `srcOr`, `srcBic`, `notSrcCopy`, and `notSrcBic` modes for

bitmaps and text. For all other objects drawn with QuickDraw, the PostScript LaserWriter driver supports only the `srcCopy` mode.

- There can be a small difference in glyph widths between fonts rendered on the screen and on the printer. Only the endpoints of text strings are the same.
- Only PostScript Level 2 supports color patterns that use colors other than red, green, blue, cyan, yellow, magenta, white, and black.
- The printer may print some large patterns at half size or smaller sizes, depending on its resolution.
- Polygons and smoothed polygons that result in the creation of paths larger than the limit of the PostScript printer (typically 1500 or 3000, depending on the version of PostScript) result in a PostScript error.

Although the PostScript LaserWriter printer is relatively fast, there are some techniques an application can use to ensure its maximum performance.

- Printing patterns takes time, because the bitmap for the pattern has to be built. The black-and-white patterns, and some of the gray patterns, have been optimized to use the PostScript grayscales.
- Use the `TextBegin` picture comment for text alignment. In the cases of flush left, flush right, or centered alignment, only the left, right, or center points are accurate, respectively; in the case of fully justified text, both the left and right endpoints are accurate.
- If you want to position each glyph independently, use the `LineLayoutOff` and `StringBegin` picture comments. If you are trying to position glyphs and the driver is trying to position glyphs too, there is conflict, and printing takes much longer than necessary.

For more information on the PostScript language, see the *PostScript Language Reference Manual*, second edition, available from Addison-Wesley.

Picture Comments to Avoid

The `SetGrayLevel` picture comment is now obsolete. The `PostScriptFile`, `TextIsPostScript`, `FormsPrinting`, `EndFormsPrinting`, `ClientLineLayout`, `PSBeginNoSave`, and `ResourcePS` picture comments have limited use and are not recommended. This section describes the shortcomings of these picture comments.

The `SetGrayLevel` picture comment was designed to provide access to the PostScript `setgray` operator while drawing with QuickDraw in black-and-white mode. For most drawing operations, however, the printer driver sets the gray level to match the foreground color for the printing graphics port, and the effect of the `SetGrayLevel` picture comment is often unpredictable. If direct access to the PostScript `setgray` operator seems desirable, it is preferable to send the instruction with the `PostScriptHandle` picture comment.

The `TextIsPostScript` picture comment interprets all the text manipulated with QuickDraw text-drawing routines (namely, `DrawChar`, `DrawString`, `DrawText`, and

anything else that calls the `StdText` low-level procedure) as PostScript code. There is no good reason to use this picture comment, but there is one important reason not to use it: printer drivers that do not support the `TextIsPostScript` picture comment will print the PostScript text instead of interpreting it. If you need to transmit PostScript code directly to a printer that understands it, use the `PostScriptHandle` comment and include a QuickDraw representation for all other printer drivers.

The `ResourcePS` picture comment loads PostScript code from a resource file. The resource file is expected to be open at the time that you use `ResourcePS`. Under background printing, there are no guarantees the resource file will still be open when the Printing Manager needs it. If you want to keep PostScript code in a resource file, it is easy to write a routine that loads the resources and sends their contents using the `PostScriptHandle` picture comment.

The `PostScriptFile` picture comment loads PostScript code from a file; as with the `ResourcePS` comment, there are no guarantees the file will be open when the Printing Manager needs it during background printing. If you want to keep PostScript code in a file, it is easy to write a routine that loads the file and its contents using the `PostScriptHandle` picture comment.

As with the `PostScriptBegin` picture comment, the `PSBeginNoSave` picture comment allows applications to change the state of a PostScript printer driver. Some applications do not want to restore the previous state of the PostScript interpreter after sending PostScript code; the `PSBeginNoSave` comment was intended for situations where applications do not want to preserve the printer state. However, the `PSBeginNoSave` picture comment allows applications to interfere with the LaserWriter 8.0 printer driver, and the driver, by calling the PostScript operator `grestore`, can interfere with the application. The use of `PSBeginNoSave` can lead to incorrect clipping, incorrect colors, and PostScript language errors and should therefore be avoided.

By default, most drivers apply about 80 percent of the total line layout error to the major glyphs (the space character) and the other 20 percent to the minor glyphs (all other glyphs). (When using a script system that does not use the space glyph to delimit words, the layout error is distributed evenly across all characters in the font.) The `ClientLineLayout` picture comment allows applications to redefine the major glyph, and the percentages of the line layout error assigned to the major and minor glyphs. The `ClientLineLayout` picture comment is rather subtle and very specific to the PostScript LaserWriter driver. Only very ambitious page layout applications might be interested in this functionality, however; their designers should instead aim at a more general scheme of line layout control that does not rely upon this very driver-specific picture comment.

Intended for printing forms on PostScript LaserWriter printers, the `FormsPrinting` picture comment directs the PostScript LaserWriter driver not to clear its page buffer after printing a page. The `EndFormsPrinting` picture comment directs the PostScript LaserWriter driver to clear its page buffer after printing a page. When a page is completed, applications must erase the areas that need to be updated and draw the new information. The graphics that make up the form are drawn only once per page, which may improve performance. However, you need to write a separate printing loop for the PostScript LaserWriter driver if you want to use this comment.

Including Constants and Data Types for Picture Comments

For the picture comments described in this appendix, neither QuickDraw nor the Printing Manager includes constant definitions or data type declarations; instead, you must include these in your own build files. Listed here are the constants and data types for picture comments that have been predefined for printer drivers from Apple Computer, Inc.

```
{PicComments.p}
CONST
   {values for picture comments}
   TextBegin         = 150;
   TextEnd           = 151;
   StringBegin       = 152;
   StringEnd         = 153;
   TextCenter        = 154;
   LineLayoutOff     = 155;
   LineLayoutOn      = 156;
   ClientLineLayout  = 157;   {considered to be of limited usefulness}
   PolyBegin         = 160;
   PolyEnd           = 161;
   PolyIgnore        = 163;
   PolySmooth        = 164;
   PolyClose         = 165;
   DashedLine        = 180;
   DashedStop        = 181;
   SetLineWidth      = 182;
   PostScriptBegin   = 190;
   PostScriptEnd     = 191;
   PostScriptHandle  = 192;
   PostScriptFile    = 193;   {considered to be of limited usefulness}
   TextIsPostScript  = 194;   {considered to be of limited usefulness}
   ResourcePS        = 195;   {considered to be of limited usefulness}
   PSBeginNoSave     = 196;   {dangerous to use with LaserWriter 8.0}
   SetGrayLevel      = 197;   {this comment now obsolete}
   RotateBegin       = 200;
   RotateEnd         = 201;
   RotateCenter      = 202;
   {values for the tJus field of the TTxtPicRec record}
   tJusNone          = 0;
   tJusLeft          = 1;
   tJusCenter        = 2;
```

```
   tJusRight         = 3;
   tJusFull          = 4;
   {values for the tFlip field of the TTxtPicRec record}
   tFlipNone         = 0;
   tFlipHorizontal   = 1;
   tFlipVertical     = 2;

TYPE
   TTxtPicHdl = ^TTxtPicPtr;
   TTxtPicPtr = ^TTxtPicRec;
   TTxtPicRec = PACKED RECORD
      tJus:          Byte;       {justification for line layout of text}
      tFlip:         Byte;       {horizontal or vertical flipping}
      tAngle:        Integer;    {0..360 degrees clockwise rotation }
                                 { in integer format}
      tLine:         Byte;       {reserved}
      tCmnt:         Byte;       {reserved}
      tAngleFixed:   Byte;       {0..360 degrees clockwise rotation in }
                                 { fixed-number format}
   END;

   TRotationHdl = ^TRotationPtr;
   TRotationPtr = ^TRotationRec;
   TRotationRec = RECORD
      rFlip:         Integer;    {horizontal/vertical flipping}
      rAngle:        Integer;    {0..360 degrees clockwise rotation }
                                 { in integer format}
      rAngleFixed:   Fixed;      {0..360 degrees clockwise rotation in }
                                 { fixed-number format}
   END;

   TCenterHdl = ^TCenterPtr;
   TCenterPtr = ^TCenterRec;
   TCenterRec = RECORD
      y:    Fixed;   {vertical offset from current pen location}
      x:    Fixed;   {horizontal offset from current pen location}
   END;

   TPolyVerbHdl = ^TPolyVerbPtr;
   TPolyVerbPtr = ^TPolyVerbRec;
```

```
TPolyVerbRec = PACKED RECORD
   f7, f6, f5, f4, f3:  Boolean; {reserved; set to 0}
   fPolyClose:          Boolean; {TRUE is same as PolyClose }
                                 { picture comment}
   fPolyFill:           Boolean; {TRUE means fill polygon}
   fPolyFrame:          Boolean; {TRUE means frame polygon}
END;

TDashedLineHdl = ^TDashedLinePtr;
TDashedLinePtr = ^TDashedLineRec;
TDashedLineRec = PACKED RECORD
   offset:     SignedByte;    {offset into pattern for first dash}
   centered:   SignedByte;    {reserved; set to 0}
   intervals:  ARRAY[0..5] OF SignedByte;
                              {points for drawing and not drawing dashes}

TLineWidthHdl = ^TLineWidthPtr;
TLineWidthPtr = ^TLineWidth;
TLineWidth = Point;  {v = numerator, h = denominator}
END;
```

Glossary

arc A portion of the circumference of an **oval,** not including the bounding radii or any part of the oval's interior.

arithmetic transfer mode A specification for how QuickDraw should draw or copy color images into a bitmap or pixel map. Arithmetic modes perform add, subtract, and blend operations on the red, green, and blue component values of RGB colors.

background color The color of the pixels wherever no drawing has taken place. By default, the background color is white.

background pattern The pattern displayed in a graphics port when an area is erased or when pixels are scrolled out of it.

background printing A feature supported by some printer drivers that allows the user to work with an application while documents are printing. These printer drivers send printing data to a spool file in the PrintMonitor Documents folder inside the System Folder.

basic graphics port The drawing environment provided by **basic QuickDraw.** A basic graphics port is defined by a data structure of type `GrafPort` and contains the information that basic QuickDraw uses to create and manipulate onscreen either black-and-white images or color images that employ the **eight-color system.**

basic QuickDraw The set of QuickDraw routines that you use to create and manipulate graphics information in a graphics port. All Macintosh computers have basic QuickDraw routines in ROM. See also **Color QuickDraw.**

bit image A collection of bits in memory that forms a grid—that is, a rectangular pattern of bits. The bit image is pointed to in the `baseAddr` field of a `BitMap` record. Compare **pixel image.**

bitmap A data structure of type `BitMap` that represents the positions and states of a corresponding set of pixels, which can be either black and white or the eight predefined colors provided by basic QuickDraw. A bitmap is contained within a **basic graphics port.** See also **pixel map.**

bit pattern An 8-by-8 pixel image drawn by default in black and white, although any two colors can be used on a color screen. A bit pattern can be repeated indefinitely to form a repeating design (such as stripes) when drawing lines and shapes or when filling areas on the screen. See also **pixel pattern.**

Boolean transfer mode A specification of which Boolean operation QuickDraw should perform when drawing or copying an image into a bitmap or pixel map. Boolean transfer modes that draw patterns are called **pattern modes;** Boolean transfer modes that copy images or draw text are called **source modes.** Compare **arithmetic transfer mode.**

boundary rectangle A rectangle (by default, the entire main screen) that links the local coordinate system of a graphics port to QuickDraw's global coordinate system and defines the area of the pixel image or bit image into which QuickDraw can draw. The boundary rectangle is stored in either the pixel map or the bitmap.

bounding rectangle A rectangle used to define other shapes, such as ovals and rounded rectangles. The lines of bounding rectangles completely enclose the shapes they bound; in other words, no pixels from these shapes lie outside the infinitely thin lines of the bounding rectangles.

clipping region A region to which an application can limit drawing. The initial clipping region of a graphics port is an arbitrarily large rectangle: one that covers the entire QuickDraw coordinate plane. An application can set the clipping region to any arbitrary region, to aid in drawing inside the graphics port.

CLUT See **color lookup table.**

color bank A structure into which all the colors of a picture, pixel map, or bitmap are gathered by the Picture Utilities or by your application for later selection. The Picture Utilities generate a color bank consisting of a **histogram** to a resolution of 5 bits per color.

color graphics port The sophisticated color drawing environment provided by **Color QuickDraw.** A color graphics port is defined by a data structure of type `CGrafPort` and contains the information that Color QuickDraw uses to create and manipulate grayscale and color images onscreen.

colorize To use the `CopyBits` procedure to copy colors into black-and-white images.

color lookup table (CLUT) A data structure that maps color indexes specified with QuickDraw into actual color values. Color lookup tables are internal to certain types of **graphics devices.** Compare **color table.**

Color Manager A set of system software routines that supply color-selection support for **Color QuickDraw.** Most applications never need to call the Color Manager directly.

Color Picker Utilities A set of system software routines that enable your application to solicit color choices from users. The Color Picker Utilities also provide routines that allow your application to convert colors between those specified in `RGBColor` records as used by Color QuickDraw and those used in other color models, such as the CMYK model used by most color printers.

Color QuickDraw The set of QuickDraw routines that you use to create and manipulate graphics information in a **color graphics port.** You can use Color QuickDraw to create a color image and then display it on any type of screen—black and white, color, or grayscale. Most Color QuickDraw routines are in ROM on Macintosh computers that use an MC68020 or faster processor. See also **basic QuickDraw.**

ColorSync Utilities A set of system software routines and algorithms that assist you in matching colors between screens and input and output devices such as scanners and printers.

color table A collection of colors available for a pixel image on indexed devices. Color tables are specified by either `ColorTable` records or `'clut'` resource types. The Color Manager stores a color table for the currently available colors in the graphics device's CLUT. Compare **color lookup table.**

current device The graphics device on which drawing is actually taking place. A handle to its `GDevice` record is stored in the global variable `TheGDevice`.

current printer The printer that the user last selected from the Chooser.

cursor A 256-bit image defined by a 16-by-16 bit square. The mouse driver displays the cursor on the screen and maps the movement of the mouse to relative locations on the screen as the user moves the mouse. The cursor follows the movement of the mouse or shows where the user's next action will take place. The cursor can be an arrow, an I-beam, a crossbar, a wristwatch, or another appropriate image. Called the pointer in Macintosh user documentation. See also **insertion point.**

Cursor Utilities A collection of system software routines for creating and using **cursors,** including color and animated cursors.

data fork The part of a file that contains data accessed using the File Manager. This data usually corresponds to data entered by the user. Compare **resource fork.**

deferred printing A method of printing whereby some printer drivers record each page of a document's printed image in a structure similar to a QuickDraw picture, which the driver writes to a spool file. An application must use the `PrPicFile` procedure to send the spool file to the printer. Deferred printing is also known as *spool printing*. Compare **draft-quality printing.**

device list A linked list containing the `GDevice` records for a user's computer system. The global variable `DeviceList` holds a handle to the first record in the list.

direct colors Up to 16 million colors that have a direct correlation between a value placed in a graphics device and the color displayed onscreen.

direct device A plug-in video card, a video interface built into a Macintosh computer, or an offscreen graphics world that supports up to 16 million colors having a direct correlation between a value placed in the device and the color displayed onscreen. Compare **indexed device.**

direct pixel A pixel displayed on a **direct device.** Direct pixels can have **pixel values** of 16 or 32 bits.

discrete resolution A printing resolution that has been predefined by a printer driver. A printer supporting discrete resolution prints only a limited number of such resolutions. Compare **variable resolution.**

dithering A technique for mixing existing colors together to create the illusion of a third color that may be unavailable on a particular device.

dpi Dots per inch in the x and y directions; used to measure the resolution of a screen or printer. The higher the value, the finer the detail of the image.

draft-quality printing The method by which printer drivers convert into drawing operations calls only to QuickDraw's text-drawing routines. The printer driver sends these routines directly to the printer instead of using **deferred printing** to capture the entire image for a page in a spool file. Draft-quality printing, which is supported on the ImageWriter printer driver, produces quick, low-quality drafts of text documents that are printed straight down the page, from top to bottom and left to right. Compare **enhanced draft-quality printing.**

eight-color system The eight predefined colors provided by **basic QuickDraw** for display on color screens and color printers.

enhanced draft-quality printing The method by which some printer drivers print bitmaps, pixel maps, and text without writing to or reading from a spool file. The ImageWriter printer driver, for example, supports enhanced draft-quality printing. Compare **deferred printing, draft-quality printing.**

erase To draw both the outline of a shape and its interior with the background pattern for the current graphics port. The background pattern is typically solid white on a black-and-white screen or a solid background color on a color screen. Making the shape blend into the background pattern of the graphics port effectively erases the shape.

extended version 2 picture format The format for all pictures created with the `OpenCPicture` function. Available on all Macintosh computers running System 7, this format allows applications to specify resolutions when creating images.

fill To draw both the outline of a shape and its interior with any pattern you specify. The procedure transfers the pattern with the `patCopy` pattern mode, which directly copies your requested pattern into the shape.

font substitution Substitution of a screen font for a printer font by a printer driver. PostScript printer drivers may substitute PostScript printer fonts for bitmapped screen fonts.

foreground color The color of the "ink" used for bit patterns and for the graphics pen when drawing. By default, the foreground color is black.

frame To draw the outline of a shape (such as a rectangle) using the size, pattern, and pattern mode of the graphics pen for the current graphics port. The interior of the shape is unaffected, allowing previously existing pixels in the image to show through.

`GDevice` record A data structure of type `GDevice` that holds information about the physical characteristics of a video device or offscreen graphics world, including a pixel map that describes the pixel depth for that video device or offscreen graphics world, information about whether the video device or offscreen graphics world supports indexed or direct colors, and—for indexed devices—specifications for the colors that are currently available for the video device or offscreen graphics world. System software allocates and initializes one `GDevice` record for each installed video device and stores the record in the system's **device list.**

global coordinate system The coordinate system that represents all potential QuickDraw drawing space. The origin of the global coordinate system—that is, the point (0,0)—is at the upper-left corner of the **main screen.** Compare **local coordinate system.**

glyph The distinct representation of a character in a form that a screen or printer can display. A glyph may represent one character (the lowercase *a*), more than one character (the *fi* ligature, two characters but one glyph), or a nonprinting character (the space character).

graphics device Anything into which QuickDraw can draw. There are three types of graphics devices: video devices (such as plug-in video cards and built-in video interfaces) that control screens, offscreen graphics worlds (which allow your application to build complex images off the screen before displaying them), and printing graphics ports. For a video device or an offscreen graphics world, Color QuickDraw stores state information in a **`GDevice` record.**

graphics pen A metaphorical device for performing drawing operations onscreen. Your application can set this pen to different sizes, patterns, and colors.

graphics port A drawing environment, defined by a `GrafPort` record (**basic graphics port**) or `CGrafPort` record (**color graphics port**), that contains all the information QuickDraw needs to transmit drawing operations from bits in memory to onscreen pixels.

gray region The region that represents all available desktop area—that is, a collection of rounded rectangles representing the display areas of all screens available to a computer.

hairlines Printed lines that are less than 1/72 of an inch wide.

highlighting A QuickDraw capability that displays background bits or pixels in a distinctive visual way, such as inverting them.

high-quality printing Printing that produces documents using all of the fonts and formatting that the user has included.

histogram A **color bank** composed of frequency counts of each color within a picture, pixel map, or bitmap at a particular resolution.

hot spot The portion of the cursor that must be positioned over a screen object before mouse clicks can have an effect on that object. Designated as a point (not a bit) in the image of the cursor. The mouse driver uses the hot spot to align the cursor with the mouse location.

idle procedure A routine that handles events and updates information while system software completes a task. For example, applications displaying a print status dialog box while a printer driver directs output to a printer typically use an idle procedure that checks for user-generated events indicating that the user wishes to cancel the printing.

imaging The construction and display of graphical information. Such graphical information can consist of shapes, pictures, and text and can be displayed on output devices such as screens and printers.

indexed colors A set of up to 256 colors contained in a video data interface called a **color lookup table** (or, more commonly, a CLUT). Video devices and offscreen graphics worlds that use indexed colors support pixels of 1-bit, 2-bit, 4-bit, or 8-bit depths.

indexed device A plug-in video card, a video interface built into a Macintosh computer, or an offscreen graphics world that supports up to 256 colors in a **color lookup table.** Indexed devices support pixels of 1-bit, 2-bit, 4-bit, or 8-bit depths. Compare **direct device.**

indexed pixel A pixel displayed on an **indexed device.** Indexed pixels can have **pixel values** of 1, 2, 4, or 8 bits.

insertion point The position where text will be inserted, usually marked by a blinking vertical bar.

inverse table A special data structure arranged by the Color Manager in such a manner that, given an arbitrary RGB color, the Color Manager can very rapidly look up its pixel value.

invert To reverse the colors of all pixels within a shape. On a black-and-white screen, this changes all the black pixels in the shape to white and all the white pixels to black. Inverting operates on color pixels in color graphics ports, but the results are predictable only with direct pixels.

job dialog box A dialog box—usually displayed by an application in response to the user choosing the Print command—that solicits printing information from the user, such as the number of copies to print, the print quality, and the range of pages to print.

line A graphic image defined by two points: the current location of the **graphics pen** and its destination. The graphics pen, which can draw with different patterns, hangs below and to the right of the defining points.

line layout error The difference between the width of the printed line and the width of the screen line after the printer driver has performed **font substitution.** Certain printer drivers compensate for this by distributing the error to **major glyphs** and **minor glyphs.**

local coordinate system The coordinate system defined by the **port rectangle** of a graphics port. When the Window Manager creates a window, it places the origin of the local coordinate system at the upper-left corner of the window's port rectangle. Compare **global coordinate system.**

luminance The intensity of light in a color. Color QuickDraw uses a color's luminance to convert the color to an appropriate grayscale color.

main screen In a drawing environment with multiple screens, the screen with the menu bar. QuickDraw maps the (0,0) origin point of the coordinate plane to the main screen's upper-left corner, and other screens are positioned adjacent to it. Compare **startup screen.**

major error The amount of line layout error that a printer driver applies to the space glyph.

major glyph On a printed page, a space glyph, to which printer drivers apply most of the **line layout error.** Compare **minor glyph.**

minor error The amount of line layout error that a printer driver applies to nonspace glyphs.

minor glyph On a printed page, a nonspace glyph, to which printer drivers apply the **line layout error** that remains after applying most of the error to **major glyphs.**

offscreen graphics world A sophisticated environment for preparing complex color or black-and-white images before displaying them on the screen. An offscreen graphics world is defined in a private data structure referred to by a pointer of type `GWorldPtr`.

opcode A value passed to a routine, such as the `DrawPicture` or `PrGeneral` procedure, that determines how the routine should operate.

oval A circular or elliptical shape defined by the bounding rectangle that encloses it. The oval is completely enclosed within the infinitely thin lines of its bounding rectangle, and never includes any pixels lying outside the bounding rectangle. If the bounding rectangle is square (that is, has equal width and height), then the oval is a circle.

page rectangle The rectangle marking the boundaries of the printable area on a page. The upper-left corner of the page rectangle always has the coordinates (0,0). The coordinates of the lower-right corner give the maximum page height and width attainable on the given printer; these coordinates are specified by the units used to express the resolution of the printing graphics port. For example, the lower-right corner of a page rectangle used by the PostScript LaserWriter printer driver for an 8.5-by-11-inch U.S. letter page is (730,552) at 72 dpi.

paint To draw the outline of a shape and its interior with the pattern of the graphics pen, using the pattern mode of the graphics pen**.**

Palette Manager A set of system software routines that allows your application to specify the colors that it needs on a window-by-window basis. The Palette Manager makes the colors available (within application-determined ranges) in a graceful manner.

paper rectangle The rectangle that describes the size of a piece of paper on which a page is printed. This rectangle is defined in the same coordinate system as the **page rectangle.** Thus, the upper-left coordinates of the paper rectangle are typically negative and its lower-right coordinates are greater than those of the page rectangle.

pattern An image that can be repeated indefinitely to form a repeating design when drawing lines and shapes or when filling areas on the screen. See also **bit pattern, pixel pattern.**

pattern mode A specification of which Boolean operation QuickDraw should perform when drawing patterns into bitmaps or pixel maps. See also **source mode.**

pen See **graphics pen.**

picture A saved sequence of QuickDraw drawing commands (and, optionally, **picture comments**) that your application can play back later with the `DrawPicture` procedure; also, the image resulting from these commands.

picture comment A command or data used for special processing by output devices, such as printer drivers. Picture comments are usually stored in the definition of a picture or are included in the code an application sends to a printer driver.

picture opcode A number that the `DrawPicture` procedure uses to determine what object to draw or what mode to change for subsequent drawing.

Picture Utilities A set of system software routines for extracting information—such as pixel depth and colors—in pictures and pixel maps.

pixel Short for *picture element*, the smallest dot that QuickDraw can draw; also, the visual representation of that dot on the screen. On a black-and-white screen, each single-color phosphor dot is a pixel that represents a bit in memory—white if the bit is 0, black if it's 1. On a color screen, three phosphor dots (red, green, and blue) compose each color pixel, which represents up to 48 bits in memory. On a grayscale screen, a white phosphor dot whose intensity can vary is a pixel that usually represents 1, 2, 4, or 8 bits in memory.

pixel depth The number of bits per pixel in a pixel image. Pixels on indexed devices can be 1, 2, 4, or 8 bits deep. (A pixel image that is 1 bit deep is equivalent to a bit image.) Pixels on direct devices can be 16 or 32 bits deep.

pixel image A collection of pixels in memory that forms a grid—a rectangular pattern of pixels. The pixel image is pointed to in the `baseAddr` field of a `PixMap` record. Compare **bit image.**

pixel map A data structure of type `PixMap` that represents the positions and states of a corresponding set of color pixels. A handle to a pixel map is contained within a **color graphics port.** See also **bitmap.**

pixel pattern An image that can be repeated indefinitely to form a repeating design (such as stripes) or tone (such as gray) when drawing lines and shapes or when filling areas on the screen. A pixel pattern can use color at any pixel depth and can be of any width and height that's a power of 2. See also **bit pattern.**

pixel value A number used by system software and a graphics device to represent a color. The translation from the color that an application specifies in an `RGBColor` record to a pixel value is performed at the time the application draws the color. The process differs for indexed and direct devices.

point The intersection of a horizontal grid line and vertical grid line on the coordinate plane, defined by a horizontal and a vertical coordinate.

polygon A graphic shape defined by any sequence of points representing the polygon's vertices, connected by straight lines from one point to the next.

port rectangle An entry in a **graphics port** that represents the area of the graphics port available for drawing—ordinarily, the content region of a window.

PostScript printer driver A printer driver that converts each QuickDraw drawing operation into the equivalent PostScript drawing operation. The driver sends the converted drawing operations to the printer—typically, a laser printer. The printer interprets the PostScript drawing operations and renders the image, thereby off-loading image processing from the computer.

printer driver A device driver that translates QuickDraw drawing routines and sends the translated instructions and data to the current printer.

printer resource file A file containing all the resources needed to run the Printing Manager with a particular printer.

printing graphics port The printing environment defined by a `TPrPort` record, which contains a QuickDraw graphics port (either a `GrafPort` or `CGrafPort` record) plus additional information used by the printer driver and system software. An application prints text and graphics by drawing into a printing graphics port using QuickDraw drawing routines, just as if drawing on the screen.

printing loop Application-supplied code that handles printing needs, such as presenting the job dialog box and determining the range of pages to be printed.

Printing Manager A collection of system software routines that your application can use to print from the Macintosh computer to any type of connected printer.

QuickDraw A collection of system software routines that performs graphics operations on the user's screen. See also **basic QuickDraw** and **Color QuickDraw.**

QuickDraw GX A collection of graphics, typography, and printing routines that provide provides applications with sophisticated color publishing capabilities. QuickDraw GX augments the capabilities of **QuickDraw.**

QuickDraw printer driver A printer driver that renders images on the Macintosh computer and then sends the rendered images in the form of bitmaps or pixel maps to the printer, which might be a dot-matrix printer, an ink jet printer, a laser printer, or a plotter.

rectangle (1) A mathematical entity defined either by its four boundaries (upper, left, lower, and right) or by two points (the upper-left and lower-right corners). Rectangles are used to define active areas on the screen, to assign coordinate systems to graphical entities, and to specify the locations and sizes for various graphical operations. (2) A rectangular shape drawn onscreen with a QuickDraw procedure such as `FrameRect` or `PaintRect`.

region An arbitrary area or set of areas on the QuickDraw coordinate plane. The outline of a region should be one or more closed loops.

resolution The degree of detail at which a device such as a printer or a screen can display an image. Resolution is usually specified in dots per inch, or **dpi,** in the x and y directions. The higher the value, the finer the detail of the image.

resource fork The part of a file that contains the files' resources, which contain data accessed using the Resource Manager. This data usually corresponds to data—such as menu, icon, and control definitions—created by the developer, but it may also include data created by the user while the application is running. Compare **data fork.**

`RGBColor` record A data structure of type `RGBColor` used to specify a color by its red, green, and blue components, with each component defined as a 16-bit integer. Color QuickDraw compares such a 48-bit value with the colors actually available on a screen's video device at execution time and chooses the closest match.

RGB color value A value that indicates the red, green, and blue components of a color. An RGB color value is specified in an **`RGBColor` record.**

rounded rectangle A rectangle with rounded corners. The figure is defined by a bounding rectangle and the width and height of the ovals forming the corners. The corner width and corner height are limited to the width and height of the bounding rectangle itself; if they are set larger, the rounded rectangle becomes an oval.

scrap The storage area maintained by the Scrap Manager to hold the last data cut or copied by the user. The scrap can reside either in memory or on disk.

source mode A specification of which Boolean operation QuickDraw should perform when copying images or text into bitmaps or pixel maps. See also **pattern mode.**

spool file A temporary disk file used by an application to store data; generally used to save memory.

spool printing See **deferred printing.**

standard state The size and location that an application deems the most convenient for a window.

startup screen The screen on which the "happy Macintosh" icon appears. By default, the menu bar appears on the startup screen. Compare **main screen.**

style dialog box A dialog box—usually displayed by an application in response to the user choosing the Page Setup command—allowing the user to specify printing options (such as the paper size and the printing orientation) that an application needs to format the document.

`TPrint` record A data structure of type `TPrint`. A `TPrint` record contains fields that specify the Printing Manager version, information about the printer (such as its resolution in dpi), and the dimensions of the paper rectangle.

`TPrJob` record A data structure of type `TPrJob`. The `TPrJob` job record contains information about a particular print job; for instance, the first and last pages to be printed, the number of copies, and the printing method (either draft-quality or deferred).

transfer mode A specification, either Boolean or arithmetic, of how QuickDraw should draw or copy images into a bitmap or pixel map. See **arithmetic transfer mode** and **Boolean transfer mode.**

user state The size and location that the user has established for a window.

variable resolution Any printing resolution within a range bounded by maximum and minimum values. Compare **discrete resolution.**

video device A piece of hardware, such as a plug-in video card or a built-in video interface, that controls a screen.

visible region The part of a window's graphics port that's actually visible on the screen—that is, the part that's not covered by other windows.

wedge A pie-shaped segment of an **oval,** bounded by a pair of radii joining at the oval's center.

window origin The upper-left corner of a window. Usually specified with a vertical coordinate of 0 and a horizontal coordinate of 0, the window origin is the upper-left corner of the port rectangle of a graphics port and is expressed in coordinates local to the graphics port.

Index

Numerals

`0..255` data type A-4
`–128..127` data type A-4

A

`Acur` data type 8-20 to 8-21
`'acur'` resource type 8-13, 8-14, 8-36 to 8-37
`addMax` arithmetic transfer mode 4-39, 4-40
`addOver` arithmetic transfer mode 4-38, 4-40
`addPin` arithmetic transfer mode 4-38, 4-40, 4-78
`AddPt` procedure 2-52
`adMin` arithmetic transfer mode 4-39, 4-40
`alignPix` flag 6-13, 6-15, 6-25
`allDevices` flag 5-30
`allInit` flag 5-17, 5-23, 5-31, 5-36
`AllocCursor` procedure 8-27
`AllowPurgePixels` procedure 6-34 to 6-35
angles, calculating 3-57
animated cursor resources 8-13, 8-14, 8-36 to 8-37
animated cursors
  creating 8-13 to 8-15, 8-31 to 8-33
  data type for 8-20 to 8-21
  resource type for 8-36 to 8-37
  user interface guidelines for 8-5, 8-13, 8-15
`AppendDITL` procedure 9-38
Apple events 9-25 to 9-26
arcs. *See also* wedges
  defined 1-14
  drawing 3-26, 3-71 to 3-77
  framing 3-72 to 3-73
  low-level routine for drawing 3-134
arithmetic transfer modes 4-38 to 4-41, 4-78
arrow cursor 8-8, 8-9 to 8-12
`arrow` global variable 2-36, 8-18
arrow region 8-9 to 8-12

B

`BackColor` procedure 3-14, 3-124
background colors 3-124, 4-72 to 4-73, 4-80
background patterns
  in basic graphics ports 2-32
  changing 3-48 to 3-49, 4-68 to 4-69
  in color graphics ports 4-51
  defined 3-7
background printing 9-9
`BackPat` procedure 3-48 to 3-49
`BackPixPat` procedure 4-68 to 4-69
basic graphics ports. *See also* color graphics ports; offscreen graphics worlds; printing graphics ports
  bitmaps in 2-32
  bit patterns in 2-13, 2-32
  boundary rectangles in 2-32
  clipping regions 2-12 to 2-13, 2-32, 2-47 to 2-49
  closing 2-38, 2-40 to 2-41
  color pictures in 7-6 to 7-7
  colors in 2-14, 2-35, 3-14 to 3-15, 3-122 to 3-125
  compared with color graphics ports 4-5 to 4-9
  copying images between 3-32 to 3-35, 3-112 to 3-122
  copying images from offscreen graphics worlds 6-9 to 6-11
  creating 2-16 to 2-17, 2-37 to 2-40
  data type for 2-30 to 2-35
  defined 1-4
  drawing areas in 2-11 to 2-13
  getting 2-18, 2-41 to 2-42, 6-8, 6-28
  opening 2-38 to 2-39
  pattern stretching in 2-35
  pen locations in 2-33
  pen modes in 2-33
  pen patterns in 2-33
  pen sizes in 2-33
  pen visibility in 2-33
  port rectangles in 2-32
  restoring 2-18, 2-42, 6-8, 6-29
  saving 2-18, 2-41 to 2-42, 6-8, 6-28
  setting 2-18, 2-42, 6-8, 6-29
  text in 2-33 to 2-34
  visible regions 2-32
basic QuickDraw
  application-defined routines for 5-35 to 5-37
  bit patterns in 1-11
  customizations of 3-35 to 3-36, 3-129
  data structures in 2-26 to 2-35, 3-36 to 3-40, 5-15 to 5-18, 6-12 to 6-15, 7-27 to 7-29, 8-16 to 8-18, 8-20 to 8-21
  drawing with 1-10 to 1-17, 3-3 to 3-141
  graphics ports in 1-5
  initializing 2-16, 2-36 to 2-37
  printing with. *See* Printing Manager

basic QuickDraw *(continued)*
resources in 3-140 to 3-141, 5-37, 7-67 to 7-68, 8-33 to 8-34, 8-36 to 8-37
routines in 2-36 to 2-54, 3-41 to 3-139, 5-19 to 5-25, 6-16 to 6-39, 7-36 to 7-46, 8-22, 8-24 to 8-31, 8-32 to 8-33
testing for availability 2-15
Bézier splines B-25
`BitClr` procedure 4-42
bit images
in bitmaps 2-9 to 2-11, 2-29
as pixel images in offscreen graphics worlds 6-9
`BitMap` data type 2-29 to 2-30. *See also* bitmaps
bitmaps
in basic graphics ports 2-9 to 2-11, 2-32
bit images in 2-9 to 2-11, 2-29
boundary rectangles for 2-10 to 2-11, 2-30
copying images between 3-32 to 3-35, 3-112 to 3-122
data type for 2-29 to 2-30
defined 1-5
fill operations in 3-108 to 3-112
local coordinate systems for 2-11
low-level routine for copying images between 3-136
as pixel maps in offscreen graphics worlds 6-3, 6-8 to 6-9
pixels in 2-11
`BitMapToRegion` function 2-49 to 2-50
bit patterns
background 3-6, 3-48 to 3-49
in basic graphics ports 2-13, 2-32
in color graphics ports 4-23 to 4-24, 4-58 to 4-59, 4-68, 4-69, 4-13
data type for 3-40
defined 1-11
filling with 3-6
framing and painting with 3-6
of graphics pens in basic graphics ports 2-33
predefined 3-6 to 3-8
resources for 3-140 to 3-141
routines for retrieving 3-126 to 3-128
`Bits16` data type 8-16
`BitsRect` opcode A-11, A-21
`BitsRgn` opcode A-11, A-21
`BkColor` opcode A-6, A-18
`BkPat` opcode A-5, A-18
`BkPixPat` opcode A-6
black-and-white QuickDraw. *See* basic QuickDraw
`black` global variable 2-36, 3-7
`blend` arithmetic transfer mode 4-38, 4-40, 4-78
Boolean transfer modes 3-8 to 3-11, 4-32 to 4-38
boundary rectangles
in basic graphics ports 2-32
in bitmaps 2-10 to 2-11, 2-30
defined 1-7
bounding rectangles 3-11
`burstDevice` flag 5-17, 5-23, 5-31, 5-36

C

`CalcCMask` procedure 4-83 to 4-84
`CalcMask` procedure 3-111 to 3-112
`CCrsr` data type 8-18 to 8-20
`CGrafPort` data type 4-48 to 4-54. *See also* color graphics ports
`CGrafPort` records
background pattern for 4-51
clipping regions 2-12 to 2-13, 2-47 to 2-49, 4-51
closing 4-67
compared with `GrafPort` records 4-8 to 4-9
copying images between 3-112 to 3-122
copying images from offscreen graphics worlds 6-9 to 6-11
creating 4-20 to 4-21, 4-63 to 4-66
disposing of 4-21, 4-63, 4-67
getting 2-18, 2-41 to 2-42, 6-8, 6-28
opening 4-63 to 4-66
pattern stretching in 4-53
pen locations in 4-52
pen modes in 4-52
pen patterns in 4-52
pen sizes in 4-52
pen visibility in 4-52
pixel maps in 4-50
port rectangles in 4-51
in printing graphics ports 9-51
restoring 2-18, 2-42, 6-8, 6-29
saving 2-18, 2-41 to 2-42, 6-8, 6-28
setting 2-18, 2-42, 6-8, 6-29
text in 4-53
visible regions 4-51
`ChExtra` opcode A-6
`'cicn'` resource type 4-105 to 4-106
classic QuickDraw. *See* basic QuickDraw
`ClientLineLayout` picture comment B-5, B-41
`Clip` opcode A-5, A-18
clipping regions 2-12 to 2-13, 2-32, 2-47 to 2-49, 4-51
`clipPix` flag 6-14, 6-15, 6-24, 6-25
`ClipRect` procedure 2-49, 3-29, 7-12
`CloseCPort` procedure 4-67
`ClosePicture` procedure 7-11, 7-42
`ClosePoly` procedure 3-79
`ClosePort` procedure 2-40 to 2-41
`CloseRgn` procedure 3-28, 3-89
`CloseWindow` procedure 7-20
CLUT. *See* color lookup tables
`'clut'` resource type 4-104 to 4-105
`CMBeginProfile` picture comment B-7

CMDisableMatching picture comment B-7
CMEnableMatching picture comment B-7
CMEndProfile picture comment B-7
'cmpt' resource type 7-68
color banks 7-33, 7-61 to 7-66
ColorBit procedure 3-124 to 3-125
color cursor resources 8-34 to 8-36
color cursors
  data structure for 8-18 to 8-20
  displaying 8-25 to 8-27
  resource for 8-34 to 8-36
  user interface guidelines for 8-5
color graphics ports. *See also* basic graphics ports; offscreen graphics worlds; printing graphics ports
  background pattern for 4-51
  clipping regions 2-12 to 2-13, 2-47 to 2-49, 4-51
  closing 4-67
  compared with basic graphic ports 4-5 to 4-9
  copying images between 3-112 to 3-122, 4-26 to 4-32
  copying images from offscreen graphics worlds 6-9 to 6-11
  creating 4-20 to 4-21, 4-63 to 4-66
  data type for 4-48 to 4-54
  defined 1-5
  disposing of 4-21, 4-63, 4-67
  getting 2-18, 2-41 to 2-42, 6-8, 6-28
  opening 4-63 to 4-66
  pattern stretching in 4-53
  pen locations in 4-52
  pen modes in 4-52
  pen patterns in 4-52
  pen sizes in 4-52
  pen visibility in 4-52
  pixel maps in 4-50
  port rectangles in 4-51
  restoring 2-18, 2-42, 6-8, 6-29
  saving 2-18, 2-41 to 2-42, 6-8, 6-28
  setting 2-18, 2-42, 6-8, 6-29
  text in 4-53
  visible regions 4-51
color icon resources 4-105 to 4-106
color lookup tables (CLUTs)
  and the Color Manager 1-24
  and the Palette Manager 1-20
  in video devices 1-19 to 1-20
Color Manager 1-29
  direct colors, handling 1-25
  indexed colors, handling 1-24
Color Picker Utilities 1-29
color-picking method resources 7-68
Color QuickDraw. *See also* global coordinate systems; local coordinate systems; shapes
  application-defined routines for 4-101 to 4-102, 5-35 to 5-37
  checking for, when zooming windows 5-10
  customizations of 3-129, 4-96 to 4-97
  data structures in 4-45 to 4-62, 5-15 to 5-18, 6-12 to 6-15, 7-27 to 7-29, 8-18 to 8-20
  direct colors, handling 1-25, 4-15 to 4-17
  drawing with 1-10 to 1-17, 4-21 to 4-44, 4-70 to 4-79
  graphics ports in 1-5
  indexed colors, handling 1-24, 4-13 to 4-14
  initializing 4-19
  multiple graphics device support in 1-21 to 1-23
  pixel patterns in 1-11
  printing with. *See* Printing Manager
  resources in 4-102 to 4-106, 5-37, 7-67 to 7-68, 8-34 to 8-36
  routines in 4-63 to 4-97, 5-19 to 5-25, 6-16 to 6-39, 8-25 to 8-27
  testing for availability 4-18
  32-bit 1-4
  user interface guidelines for 4-44
  versions of 1-4
colors
  application-defined picking method 7-61 to 7-67
  in basic graphics ports 2-14, 2-35, 3-14 to 3-15
  in color graphics ports 4-67 to 4-105
  determining 4-79 to 4-81, 7-26
  on grayscale devices 4-17
  intermediate 4-81
color search functions 4-101 to 4-102
ColorSpec data type 4-55 to 4-56
ColorSync Utilities 1-29
ColorTable data type 4-56 to 4-57. *See also* color tables
color table resources 4-104 to 4-105
color tables. *See also* color lookup tables
  creating 4-92 to 4-93, 4-104 to 4-105
  data type for 4-56 to 4-57
  default 4-93
  defined 4-11 to 4-12
  disposing of 4-93
  modifying 4-97 to 4-98
  resource type for 4-104 to 4-105
CommentSpec data type 7-30
content areas of windows. *See* port rectangles
coordinate planes 1-6 to 1-10. *See also* global coordinate systems; local coordinate systems
copies, to print 9-19
CopyBits procedure 3-32 to 3-34, 3-112 to 3-118, 4-26 to 4-28, 6-6, 6-9
CopyDeepMask procedure 3-120 to 3-122, 4-30 to 4-32, 6-10
CopyMask procedure 3-119 to 3-120, 4-28 to 4-30, 6-10 to 6-11
CopyPixMap procedure 4-86
CopyPixPat procedure 4-90
CopyRgn procedure 3-90 to 3-91, 8-11

CQDProcs data type 4-60 to 4-61
crosshairs cursor 8-8 to 8-9
'crsr' resource type 8-34 to 8-36
cSpecArray data type 4-55 to 4-56
CTabChanged procedure 4-97 to 4-98
current device
  defined 5-4
  determining 5-26
  setting 5-24
current printer
  defined 9-3
  device number of 9-48
  feed type of 9-48
Cursor data type 8-16 to 8-18
cursor resources 8-13 to 8-14, 8-33 to 8-34
cursors
  animating 8-13 to 8-15, 8-31 to 8-33
  arrow 8-8, 8-9 to 8-12
  changing 8-7 to 8-13, 8-26 to 8-27
  color 8-18 to 8-20, 8-25 to 8-27, 8-34 to 8-36
  crosshairs 8-8 to 8-9
  data types for 8-16 to 8-21
  defined 8-3 to 8-4
  getting from resources 8-24, 8-26
  hiding 8-28 to 8-29
  hot spots for 8-19
  I-beam 8-8 to 8-12
  initializing 8-6 to 8-7, 8-21 to 8-23
  obscuring 8-29
  plus sign 8-8 to 8-9
  resources for 8-33 to 8-37
  setting the appearance of 8-7
  shielding behind rectangles 8-29
  showing, after hiding 8-30 to 8-31
  user interface guidelines for 8-4 to 8-5
  wristwatch 8-8 to 8-9
Cursors data type 8-20
Cursor Utilities 8-3 to 8-43
  data structures in 8-16 to 8-21
  resources for 8-33 to 8-37
  routines in 8-21 to 8-33
'CURS' resource type 8-13 to 8-14, 8-33 to 8-34

D

DashedLine picture comment B-6, B-9, B-33 to B-35
dashed lines B-33 to B-35
DashedStop picture comment B-6, B-9, B-34
data forks 7-7
DCE (device control entry), for printer drivers 9-80 to 9-81
deferred printing 9-24, 9-71 to 9-72
DefHilite opcode A-7
DeltaPoint function 2-53
destination rectangles, for the DrawPicture procedure 7-18 to 7-19
device control entry, for printer drivers 9-80 to 9-81
DeviceList global variable 5-4
device lists
  defined 5-4
  getting first device in 5-26 to 5-27
DeviceLoopFlags data type 5-18 to 5-19
DeviceLoop procedure 5-8 to 5-9, 5-29 to 5-30
DHDVText opcode A-7, A-19
DHText opcode A-7, A-19
dialog boxes, for printing. *See also* job dialog boxes; print status dialog boxes; style dialog boxes
  altering 9-35 to 9-38, 9-63 to 9-66, 9-86
  data structure for 9-50 to 9-51
  displaying 9-13 to 9-15, 9-61 to 9-64
dialog hooks 9-37, 9-38
Dialog Manager
  and Printing Manager 9-5 to 9-8, 9-35 to 9-38
  and QuickDraw 4-6
diameters of curvature 1-14
DiffRgn procedure 3-96, 8-11
DirectBitsRect opcode A-11
DirectBitsRgn opcode A-12
direct colors 1-19, 1-20, 1-25
direct devices
  defined 4-5
  pixel values for 4-15 to 4-17
discrete resolution 9-11, 9-30 to 9-32
DisposCCursor procedure. *See* DisposeCCursor procedure
DisposCTable procedure. *See* DisposeCTable procedure
DisposeCCursor procedure 8-27
DisposeCTable procedure 4-93
DisposeGDevice procedure 5-25
DisposeGWorld procedure 6-6, 6-26 to 6-27
DisposePictInfo function 7-60
DisposePixMap procedure 4-87
DisposePixPat procedure 4-25, 4-91
DisposeRgn procedure 3-28, 3-90
DisposeScreenBuffer procedure 6-27
DisposeWindow procedure 7-13, 7-20
DisposPictInfo function. *See* DisposePictInfo function
DisposPixMap procedure. *See* DisposePixMap procedure
DisposPixPat procedure. *See* DisposePixPat procedure
ditherCopy mode 4-37
dithering 4-37 to 4-38
ditherPix flag 6-14, 6-15, 6-24, 6-25
dkGray global variable 2-36, 3-7 to 3-8

documents
names for, when printing 9-27
printing 9-18 to 9-26, 9-66 to 9-72
dontMatchSeeds flag 5-30
draftBitsOp opcode 9-33 to 9-35, 9-52, 9-55
draft-quality printing 9-24, 9-55. *See also* enhanced draft-quality printing
DrawPicture procedure 7-12, 7-18 to 7-19, 7-44 to 7-45
DVText opcode A-7, A-19

E

eight-color system 3-14 to 3-15, 3-122 to 3-125
EmptyRect function 3-58
EmptyRgn function 3-99
EndFormsPrinting picture comment B-7, B-41
EndofPicture opcode A-21
enhanced draft-quality printing 9-33 to 9-35, 9-55, 9-73
EqualPt function 2-54
EqualRect function 3-58
EqualRgn function 3-98
eraseArc opcode A-9, A-20
EraseArc procedure 3-76
eraseOval opcode A-9, A-20
EraseOval procedure 3-70
erasePoly opcode A-10, A-20
ErasePoly procedure 3-84
eraseRect opcode A-8, A-19
EraseRect procedure 3-61 to 3-62, 4-35, 5-10, 6-11
eraseRgn opcode A-10, A-21
EraseRgn procedure 3-102 to 3-103
EraseRoundRect procedure 3-66 to 3-67
eraseRRect opcode A-8, A-19
eraseSameArc opcode A-10, A-20
eraseSameOval opcode A-9, A-20
eraseSamePoly opcode A-10, A-21
eraseSameRect opcode A-8, A-19
eraseSameRgn opcode A-11, A-21
eraseSameRRect opcode A-8, A-19
erasing shapes 3-12
error handling
for Color QuickDraw routines 4-94 to 4-95
for printing 9-41 to 9-42, 9-73, 9-75 to 9-78
event filter functions 9-36, 9-38
ext32Device flag 5-17, 5-23, 5-31, 5-36
extended version 2 format 7-5 to 7-6, 7-37 to 7-39, A-3, A-5 to A-14, A-23 to A-24

F

feed types 9-48
FgColor opcode A-6, A-18
File menu
Page Setup command 9-5 to 9-7
Print command 9-5 to 9-6, 9-7 to 9-8
fillArc opcode A-9, A-20
FillArc procedure 3-75
FillCArc procedure 4-76
FillCOval procedure 4-75
FillCPoly procedure 4-76 to 4-77
FillCRect procedure 4-25, 4-74
FillCRgn procedure 4-77
FillCRoundRect procedure 4-74 to 4-75
filling shapes 3-12, 3-108 to 3-112
fillOval opcode A-9, A-20
FillOval procedure 3-69 to 3-70
FillPat opcode A-6
fill patterns
in basic graphics ports 2-32
in color graphics ports 4-74 to 4-77
FillPixPat opcode A-6
fillPoly opcode A-10, A-20
FillPoly procedure 3-30, 3-83 to 3-84
fillRect opcode A-8, A-19
FillRect procedure 3-23 to 3-24, 3-60 to 3-61, 4-22
fillRgn opcode A-11, A-21
FillRgn procedure 3-28, 3-102
FillRoundRect procedure 3-65 to 3-66
fillRRect opcode A-8, A-19
fills
calculating black-and-white 3-108 to 3-112
calculating color 4-82 to 4-84
fillSameArc opcode A-10, A-20
fillSameOval opcode A-9, A-20
fillSamePoly opcode A-10, A-21
fillSameRect opcode A-8, A-19
fillSameRgn opcode A-11, A-21
fillSameRRect opcode A-8, A-20
FindControl function 2-19
Finder, printing from 9-25 to 9-26, 9-66
Fixed data type A-4
fontName opcode A-7
FontSpec data type 7-30 to 7-32
font substitution B-11 to B-14
ForeColor procedure 3-14, 3-123
foreground colors 3-123, 3-124 to 3-125, 4-21 to 4-23, 4-70 to 4-71, 4-79

formats for pictures
  extended version 2 7-5 to 7-6, 7-37 to 7-39, A-3, A-5 to A-14, A-23 to A-24
  version 1 7-5 to 7-6, A-3 to A-4, A-5, A-18 to A-21, A-25 to A-26
  version 2 7-5 to 7-6, 7-39, A-3, A-5 to A-16, A-24 to A-25
FormsPrinting picture comment B-7, B-41
FractEnable global variable B-15
frameArc opcode A-9, A-20
FrameArc procedure 3-26, 3-72 to 3-73
frameOval opcode A-9, A-20
FrameOval procedure 3-25, 3-68
framePoly opcode A-10, A-20
FramePoly procedure 3-81 to 3-82
frameRect opcode A-8, A-19
FrameRect procedure 3-22 to 3-23, 3-59
frameRgn opcode A-10, A-21
FrameRgn procedure 3-100 to 3-101
FrameRoundRect procedure 3-64
frameRRect opcode A-8, A-19
frameSameArc opcode A-10, A-20
frameSameOval opcode A-9, A-20
frameSamePoly opcode A-10, A-20
frameSameRect opcode A-8, A-19
frameSameRgn opcode A-11, A-21
frameSameRRect opcode A-8, A-19
framing shapes 3-12
FSpOpenDF function 7-14

G

gdDevType flag 5-17, 5-23, 5-31, 5-33, 5-34, 5-36
GDeviceChanged procedure 4-100
GDevice data type 5-15 to 5-18. *See also* graphics devices
GDevice records. *See also* graphics devices
  creating 5-20 to 5-23
  disposing of 5-25
  getting available 5-25 to 5-28
  with greatest pixel depth 5-27 to 5-28
  modifying 4-100
  for multiple devices 1-21 to 1-23
  setting attributes for 5-22 to 5-23
  setting for current device 5-24
gestaltQuickDrawFeatures selector 4-19
gestaltQuickDrawVersion selector 4-18
GetBackColor procedure 4-80
GetCCursor function 8-26
GetClip procedure 2-47, 3-29
GetCPixel procedure 4-80 to 4-81
GetCTable function 4-92 to 4-93
GetCursor function 8-11, 8-24
GetDeviceList function 5-11, 5-26 to 5-27
GetForeColor procedure 4-79
GetGDevice function 5-26
GetGray function 4-81
GetGWorldDevice function 6-30
GetGWorldPixMap function 6-6, 6-31 to 6-32
GetGWorld procedure 6-6, 6-28
GetIndPattern procedure 3-127 to 3-128
GetMainDevice function 5-11, 5-27
GetMaxDevice function 5-27 to 5-28
GetNewCWindow function 2-16 to 2-17, 4-20
GetNewWindow function 2-16 to 2-17, 4-20
GetNextDevice function 5-11, 5-28
GetPattern function 3-126 to 3-127
GetPen procedure 3-43
GetPenState procedure 3-43
GetPictInfo function 7-25, 7-47 to 7-50
GetPicture function 7-46
GetPixBaseAddr function 6-38 to 6-39
GetPixel function 2-54 to 2-55
GetPixelsState function 6-36 to 6-37
GetPixMapInfo function 7-50 to 7-52
GetPixPat function 4-25, 4-88
GetPort procedure 2-18, 2-41 to 2-42
getRotnOp opcode 9-32 to 9-33, 9-52, 9-56
getRslDataOp opcode 9-30 to 9-32, 9-53 to 9-54
GetWindowPic function 7-13
global coordinate systems
  converting to local coordinate systems 2-19, 2-51
  defined 1-6 to 1-10
  across multiple screens 1-21
GlobalToLocal procedure 2-19, 2-51
global variables
  arrow 2-36, 8-18
  black 2-36, 3-7
  DeviceList 5-4
  dkGray 2-36, 3-7 to 3-8
  FractEnable B-15
  gray 2-36, 3-7
  HiliteRGB 4-42
  ltGray 2-36, 3-7
  MainDevice 5-27
  PrintErr 9-78
  QDColors 4-71
  randSeed 2-36
  screenBits 2-36
  ScrHRes 5-32
  ScrVRes 5-32
  TheGDevice 5-4
  thePort 2-36
  TopMapHdl 9-39
  white 2-36, 3-7
glyphs B-12
GrafPort data type 2-30 to 2-35. *See also* basic graphics ports

GrafPort records
bitmaps in 2-32
bit patterns in 2-13, 2-32
boundary rectangles in 2-32
clipping regions 2-12 to 2-13, 2-32, 2-47 to 2-49
closing 2-38, 2-40 to 2-41
and color pictures 7-6 to 7-7
colors in 2-14, 2-35, 3-14 to 3-15, 3-122 to 3-125
compared with CGrafPort records 4-8 to 4-9
copying images between 3-32 to 3-35, 3-112 to 3-122
copying images from offscreen graphics worlds 6-9 to 6-11
creating 2-16 to 2-17, 2-37 to 2-40
drawing areas in 2-11 to 2-13
getting 2-18, 2-41 to 2-42, 6-8, 6-28
opening 2-38 to 2-39
pattern stretching in 2-35
pen locations in 2-33
pen modes in 2-33
pen patterns in 2-33
pen sizes in 2-33
pen visibility in 2-33
port rectangles in 2-32
in printing graphics ports 9-51
restoring 2-18, 2-42, 6-8, 6-29
saving 2-18, 2-41 to 2-42, 6-8, 6-28
setting 2-18, 2-42, 6-8, 6-29
text in 2-33 to 2-34
visible regions 2-32
GrafVars data type 4-62
GrafVerb data type 3-132
graphics device records. *See* GDevice records
graphics devices 5-3 to 5-44. *See also* GDevice records
application-defined routine for 5-35 to 5-37
data structures in 5-15 to 5-18
defined 5-3
determining characteristics of 5-8 to 5-9, 5-29 to 5-32
getting handles to 5-25 to 5-28
with greatest pixel depth 5-27 to 5-28
initialization 1-22 to 1-23
optimizing images for 5-8 to 5-13, 5-29 to 5-30, 5-35 to 5-37
resource for 5-37
routines for 5-19 to 5-25
testing for availability 5-8
graphics pens
attributes of 1-11, 2-33, 3-4 to 3-5, 4-52
bit patterns for 3-20 to 3-21, 3-43, 3-48 to 3-49
colors for 3-123, 4-21 to 4-26, 4-67 to 4-68, 4-70 to 4-71
defined 1-5
drawing with 1-10 to 1-17
in graphics ports 2-13
initial values 3-48
invisible state 3-42
locations of 3-43
moving 3-17, 3-18 to 3-19, 3-50 to 3-51
pattern modes 3-43 to 3-48
pixel patterns for 4-67 to 4-68
routines for managing 3-41 to 3-48
sizes of 3-19 to 3-20, 3-43 to 3-44, 3-48
visible state 3-42
graphics port records. *See* CGrafPort records; GrafPort records; TPrPort records
graphics ports. *See also* basic graphics ports; color graphics ports; offscreen graphics worlds; printing graphics ports
background patterns in 2-32
clipping regions 2-12 to 2-13, 2-47 to 2-49
copying images between 3-32 to 3-35, 3-112 to 3-122
creating 1-7 to 1-8
data types for 2-30 to 2-35, 4-48 to 4-54, 9-51 to 9-52
defined 1-4
drawing areas in 2-11 to 2-13
fill patterns in 2-32
getting 2-18, 2-41 to 2-42
graphics pens in 2-13
local coordinate systems in 2-13
modifying 4-99 to 4-100
patterns in 2-13
port rectangles in 2-11
printing in 9-4 to 9-5, 9-15 to 9-35, 9-66 to 9-74, B-3 to B-42
restoring 2-18, 2-42, 6-8, 6-27 to 6-29
saving 2-18, 2-41 to 2-42, 6-8, 6-27 to 6-29
setting 2-18, 2-42, 6-8, 6-27 to 6-29
text in 2-13
visible regions 2-11
as windows 1-7 to 1-8
gray global variable 2-36, 3-7
grayscale devices, colors on 4-17
gwFlagErr flag 6-14
GWorld. *See* offscreen graphics worlds
GWorldFlags data type 6-13 to 6-15
GWorldPtr data type 6-12

H

hairlines B-35 to B-37
HasDepth function 5-13, 5-33 to 5-34
header information A-3
HeaderOp opcode A-3, A-13
Hide_Cursor procedure 8-28
HideCursor procedure 8-28
HidePen procedure 3-42
highlighting 4-41 to 4-44, 4-78 to 4-79
HiliteColor opcode A-7
HiliteColor procedure 4-78 to 4-79
hilite mode 4-44

HiliteMode opcode A-7
HiliteRGB global variable 4-42
histograms 7-61, 7-63 to 7-64
hot spots 8-4, 8-19

I

I-beam cursor 8-9 to 8-12
I-beam region 8-9 to 8-12
idle procedures 9-13 to 9-15, 9-21, 9-38 to 9-41, 9-85
images
  copying 3-112 to 3-122, 4-26 to 4-32, 6-9 to 6-11
  scrolling 2-20 to 2-26, 2-43 to 2-44
ImageWriter LQ printers B-7
imaging, defined 1-3
indexed colors 1-19 to 1-20, 1-24 to 1-25
indexed devices
  defined 4-5
  pixel values for 4-13 to 4-14
InitCPort procedure 4-66
InitCursorCtl procedure 8-7, 8-22 to 8-23
InitCursor procedure 8-7, 8-22
InitGDevice procedure 5-21 to 5-22
InitGraf procedure 2-36 to 2-37
initialization, of graphics system 1-22 to 1-23
InitPort procedure 2-39 to 2-40
InsetRect procedure 3-54
InsetRgn procedure 3-93 to 3-94
Integer data type A-4
inverse tables, defined 5-5
invertArc opcode A-9, A-20
InvertArc procedure 3-77
inverting shapes 3-13
invertOval opcode A-9, A-20
InvertOval procedure 3-71
invertPoly opcode A-10, A-20
InvertPoly procedure 3-85
invertRect opcode A-8, A-19
InvertRect procedure 3-62
invertRgn opcode A-10, A-21
InvertRgn procedure 3-103 to 3-104
InvertRoundRect procedure 3-67 to 3-68
invertRRect opcode A-8, A-19
invertSameArc opcode A-10, A-20
invertSameOval opcode A-9, A-20
invertSamePoly opcode A-10, A-21
invertSameRect opcode A-8, A-19
invertSameRgn opcode A-11, A-21
invertSameRRect opcode A-8, A-20

J

job dialog boxes
  altering 9-35 to 9-38, 9-63 to 9-64, 9-65, 9-86
  defined 9-6
  displaying 9-62 to 9-63
  for LaserWriter printers 9-8
  for multiple documents 9-26, 9-66
  for StyleWriter printers 9-7 to 9-8

K

keepLocal flag 6-13, 6-14, 6-18, 6-20, 6-24
KillPicture procedure 7-13, 7-42 to 7-43
KillPoly procedure 3-30, 3-80 to 3-81

L

landscape printing 9-32 to 9-33, 9-34, 9-56, 9-73
LaserWriter printers 9-7, 9-76, B-7
LaserWriter SC printers B-7
LineFrom opcode A-7, A-19
lineJustify opcode A-7
line layout, disabling and enabling B-11 to B-17
line layout error B-12 to B-16
LineLayoutOff picture comment B-5, B-15 to B-16, B-17
LineLayoutOn picture comment B-5, B-15, B-17
Line opcode A-7, A-19
Line procedure 3-18 to 3-19, 3-51 to 3-52
lines
  defined 1-12
  defining 3-11 to 3-12
  drawing 3-17 to 3-21, 3-49 to 3-52
  low-level routine for drawing 3-132
  printing, with picture comments B-33 to B-37
LineTo procedure 3-17 to 3-18, 3-51
local coordinate systems
  for bitmaps 2-11
  converting to global coordinate systems 2-19, 2-52
  defined 1-7 to 1-10
  in graphics ports 2-13
LocalToGlobal procedure 2-52
LockPixels function 6-6, 6-32 to 6-33
LongComment opcode A-12, A-21
Long data type A-4
LongText opcode A-7, A-19
ltGray global variable 2-36, 3-7
luminance 4-17

M

magic pen B-8. *See also* pattern modes
MainDevice global variable 5-27
main screen
  defined 1-21
  determining 5-27
mainScreen flag 5-17, 5-23, 5-31, 5-36
major error B-13 to B-14
major glyphs B-12 to B-14
MakeRGBPat procedure 4-90 to 4-91
mapPix flag 6-13, 6-15, 6-25
MapPoly procedure 3-108
MapPt procedure 3-106
MapRect procedure 3-106 to 3-107
MapRgn procedure 3-107
MatchRec data type 4-57
minor error B-13 to B-14
minor glyphs B-12 to B-14
–128..127 data type A-4
Mode data type A-4
mouse region 8-9 to 8-12
MovePortTo procedure 2-46 to 2-47
Move procedure 3-18 to 3-19, 3-50 to 3-51
MoveTo procedure 3-17 to 3-18, 3-50
multiple graphics devices 1-21 to 1-23
MyCalcColorTable function 7-65 to 7-66
MyColorSearch function 4-101 to 4-102
MyDisposeColorPickMethod function 7-67
MyDoPrintIdle procedure 9-85
MyDrawingProc procedure 5-36 to 5-37
MyInitPickMethod function 7-62 to 7-64
MyPrDialogAppend function 9-86
MyRecordColors function 7-64 to 7-65

N

newDepth flag 6-13, 6-15, 6-25
NewGDevice function 5-20 to 5-21
NewGWorld function 6-5 to 6-7, 6-16 to 6-21
NewPictInfo function 7-53 to 7-55
NewPixMap function 4-85 to 4-86
NewPixPat function 4-88 to 4-89
NewRgn function 3-28, 3-87
newRowBytes flag 6-13, 6-15, 6-25
NewScreenBuffer function 6-21 to 6-22
NewTempScreenBuffer function 6-22 to 6-23
noDraftBitsOp opcode 9-52, 9-55
noDriver flag 5-17, 5-23, 5-31, 5-36
noNewDevice flag 6-13, 6-14, 6-18, 6-20, 6-30
NOP opcode A-5, A-18
NoPurgePixels procedure 6-35
notPatBic pattern mode 3-9 to 3-10, 3-45
notPatCopy pattern mode 3-9 to 3-10, 3-45
notPatOr pattern mode 3-9 to 3-10, 3-45
notPatXor pattern mode 3-9 to 3-10, 3-45
notSrcBic source mode 3-9 to 3-10, 3-114, 3-116
notSrcCopy source mode 3-9 to 3-10, 3-114, 3-115, 4-33, 4-34
notSrcOr source mode 3-9 to 3-10, 3-114, 3-115, 4-33, 4-34
notSrcXor source mode 3-9 to 3-10, 3-114, 3-116, 4-33

O

ObscureCursor procedure 8-29
offscreen graphics worlds 6-3 to 6-46
  copying images from 6-9 to 6-11
  creating 6-5 to 6-7, 6-16 to 6-23
  data structures in 6-12 to 6-15
  defined 6-3
  disposing of 6-26 to 6-27
  drawing into 6-8 to 6-9
  restoring 6-8, 6-27 to 6-29
  routines for 6-16 to 6-39
  saving 6-8, 6-27 to 6-29
  setting 6-8, 6-27 to 6-29
  testing for availability 6-5
  updating 6-9, 6-23 to 6-26
OffsetPoly procedure 3-80
OffsetRect procedure 3-53 to 3-54
OffsetRgn procedure 3-93
Opcode data type A-4
opcodes 7-6
  for pictures A-3 to A-26
  for the PrGeneral procedure 9-52, 9-72 to 9-74
OpColor opcode A-7
OpColor procedure 4-78
OpenCPicParams records 7-29
OpenCPicture function 7-11, 7-37 to 7-39
OpenCPort procedure 4-64 to 4-65
OpEndPic opcode A-3, A-12
OpenPicture function 7-39 to 7-40
OpenPoly function 3-30, 3-78 to 3-79
OpenPort procedure 2-38 to 2-39
OpenRgn procedure 3-28, 3-87 to 3-88
original Color QuickDraw. *See* Color QuickDraw
Origin opcode A-6, A-18
origins. *See* window origins
ovals
  defined 1-13
  drawing 3-25, 3-68 to 3-71
  erasing 3-70
  filling
    with bit patterns 3-69 to 3-70
    with pixel patterns 4-75

ovals *(continued)*
framing 3-68
inverting 3-71
painting 3-69
and rounded rectangles 1-14
`OvSize` opcode A-6, A-18

P

`PackBitsRect` opcode A-11, A-21
`PackBitsRgn` opcode A-11, A-21
page rectangles 9-10 to 9-11, 9-46
pages
determining number to print 9-19, 9-23
orientation of 9-32 to 9-33
printable area for 9-10 to 9-11
printing 9-19 to 9-24, 9-69 to 9-70
Page Setup command (File menu) 9-5 to 9-7
`paintArc` opcode A-9, A-20
`PaintArc` procedure 3-26, 3-73 to 3-74
painting shapes 3-12
`paintOval` opcode A-9, A-20
`PaintOval` procedure 3-69
`paintPoly` opcode A-10, A-20
`PaintPoly` procedure 3-82 to 3-83
`paintRect` opcode A-8, A-19
`PaintRect` procedure 3-23 to 3-24, 3-60, 4-22, 4-25
`paintRgn` opcode A-10, A-21
`PaintRgn` procedure 3-101
`PaintRoundRect` procedure 3-64 to 3-65
`paintRRect` opcode A-8, A-19
`paintSameArc` opcode A-10, A-20
`paintSameOval` opcode A-9, A-20
`paintSamePoly` opcode A-10, A-21
`paintSameRect` opcode A-8, A-19
`paintSameRgn` opcode A-11, A-21
`paintSameRRect` opcode A-8, A-19
Palette Manager 1-20, 1-29
paper rectangles 9-10
`'PAT#'` resource type 3-127 to 3-128, 3-141
`patBic` pattern mode 3-9 to 3-10, 3-45
`patCopy` pattern mode 3-9 to 3-10, 3-45
`patOr` pattern mode 3-9 to 3-10, 3-45
`'PAT '` resource type 3-126 to 3-127, 3-140
`Pattern` data type 3-40, A-4. *See also* bit patterns
pattern list resources 3-127 to 3-128, 3-141
pattern modes 3-8 to 3-11, 4-33
changing 3-45 to 3-46
"magic," for PostScript printers B-22, B-30 to B-32, B-34
pattern resources 3-126 to 3-127, 3-140
patterns. *See also* bit patterns; pixel patterns
background, in basic graphics ports 2-32
background, in color graphics ports 4-68 to 4-69
in basic graphics ports 2-13, 2-32
changing 3-47 to 3-49, 4-68 to 4-69
data types for 3-40, 4-58 to 4-60
defined 1-11
fill, in basic graphics ports 2-32
fill, in color graphics ports 4-74 to 4-77
of graphics pens in basic graphics ports 2-33
of graphics pens in color graphics ports 4-67 to 4-68
resources for 3-140 to 3-141, 4-103
stretching for printer output 2-35, 4-53
`patXor` pattern mode 3-9 to 3-10, 3-45
`PenMode` procedure 3-45 to 3-46, B-22, B-30 to B-32, B-34
pen modes. *See* pattern modes
`PenNormal` procedure 3-48
`PenPat` procedure 3-20 to 3-21, 3-47
`PenPixPat` procedure 4-67 to 4-68
pens. *See* graphics pens
`PenSize` procedure 3-19 to 3-20, 3-44
pen state 3-37 to 3-38
`PenState` data type 3-37 to 3-38
Personal LaserWriter LS printers B-7
`PicComment` procedure 7-40 to 7-42, B-3 to B-41
`'PICT'` file type 7-7, 7-13 to 7-16, 7-21 to 7-23
`PictInfo` data type 7-32 to 7-36
`'PICT'` resource type 7-7, 7-20, 7-46, 7-67 to 7-68
`'PICT'` scrap format 7-7 to 7-8, 7-17, 7-22
picture comments 7-40 to 7-42, B-3 to B-44
defined 7-6
delimiting text strings with B-16 to B-17
device independence and printing B-8 to B-9
disabling and enabling line layout with B-11 to B-17
graphics rotation with B-29 to B-32
inserting into pictures or printing code 7-40 to 7-42
limited or obsolete B-40 to B-41
low-level routine for processing 3-137
matching colors with B-7
printing dashed lines with B-33 to B-35
printing graphics with B-6, B-22 to B-32
printing hairlines with B-35 to B-37
printing polygons with B-23 to B-29
printing ruled lines with B-6, B-33 to B-37
printing text with B-5, B-11 to B-22
sending PostScript printing code with B-6, B-38 to B-40
synchronizing between QuickDraw and PostScript printer drivers B-10 to B-11
text rotation with B-17 to B-22
`Picture` data type 7-27 to 7-28. *See also* pictures
picture opcodes A-3 to A-26
picture resources 7-7, 7-20, 7-46, 7-67 to 7-68
pictures
collecting information from 7-24 to 7-26, 7-46 to 7-50, 7-53 to 7-57, 7-58 to 7-60

color, in basic graphics ports 7-6 to 7-7
creating 7-10 to 7-13, 7-37 to 7-42
data type for 7-27 to 7-28
defined 1-16, 7-4
destination rectangles for 7-18 to 7-19
disposing of 7-13, 7-20, 7-42 to 7-43
drawing 7-10 to 7-20, 7-43 to 7-45
extended version 2 format 7-5 to 7-6, 7-37 to 7-39, A-3, A-5 to A-14, A-23 to A-24
low-level routines for 3-138 to 3-139
opcodes for 7-6
opening 7-13 to 7-20
in 'PICT' files 7-7, 7-13 to 7-16, 7-21 to 7-23
in 'PICT' resources 7-7, 7-20, 7-22, 7-46
reading from a resource file 7-46
resolutions for 7-11, 7-19
saving 7-21 to 7-23
in the scrap 7-7 to 7-8, 7-17, 7-22
version 1 format 7-5 to 7-6, A-3 to A-4, A-5, A-18 to A-21, A-25 to A-26
version 2 format 7-5 to 7-6, 7-39, A-3, A-5 to A-16, A-24 to A-25
and the Window Manager 7-13
Picture Utilities
application-defined routines for 7-61 to 7-67
data structures in 7-30 to 7-36
defined 7-8
gathering information with 7-24 to 7-26
routines in 7-46 to 7-60
testing for availability 7-10
picVersion opcode A-19
PixData data type A-4, A-15
pixel depths
default color tables for 4-93
defined 4-10
determining 5-8 to 5-13, 5-29 to 5-30, 5-33 to 5-34
setting 5-13, 5-34 to 5-35
pixel images
addresses of, for offscreen graphics worlds 6-38 to 6-39
defined 4-10 to 4-12
getting states of, for offscreen graphics worlds 6-36 to 6-37
locking, for offscreen graphics worlds 6-32 to 6-33
in pixel maps 4-10 to 4-12
purgeable, for offscreen graphics worlds 6-34 to 6-35
setting states, for offscreen graphics worlds 6-37 to 6-38
unlocking, for offscreen graphics worlds 6-33 to 6-34
unpurgeable, for offscreen graphics worlds 6-35
whether in 32-bit mode, for offscreen graphics worlds 6-39
pixel maps
copying images between 3-112 to 3-122, 4-26 to 4-32
creating 4-85 to 4-86
data type for 4-46 to 4-48
defined 1-5, 4-9
disposing of 4-87
gathering color information from 7-50 to 7-55, 7-57 to 7-60
low-level routine for copying images between 3-136
obtaining, for offscreen graphics worlds 6-31 to 6-32
pixel images in 4-10 to 4-12
setting 4-86 to 4-87
pixel pattern resources 4-24 to 4-25, 4-103
pixel patterns
background 4-68 to 4-69
creating 4-88 to 4-91, 4-103
data type for 4-58 to 4-60
defined 1-11, 4-12 to 4-13
disposing of 4-91
filling with 4-23 to 4-26, 4-74 to 4-77
framing and painting with 4-23 to 4-26
of graphics pens 4-23 to 4-26, 4-67 to 4-68
modifying 4-98 to 4-99
resources for 4-24 to 4-25, 4-103
pixels
in bitmaps 2-11
colors for
in basic QuickDraw eight-color system 3-14 to 3-15, 3-122 to 3-125
in Color QuickDraw 4-4 to 4-5, 4-10 to 4-11, 4-13 to 4-17, 4-21 to 4-44
copying between bitmaps 3-32 to 3-35, 3-112 to 3-122
copying between pixel maps 3-32 to 3-35, 3-112 to 3-122, 4-26 to 4-32
copying from offscreen graphics worlds 3-112 to 3-122, 6-9 to 6-11
defined 1-4
depths of. *See* pixel depths
patterns for. *See* bit patterns, pixel patterns
relationship to points 1-9
scrolling 2-20 to 2-26, 2-43 to 2-44
values for. *See* pixel values
whether black or white 2-54 to 2-55
whether in rectangles 3-56
whether in regions 3-97
pixelsLocked flag 6-13, 6-15, 6-36, 6-37
pixelsPurgeable flag 6-13, 6-14, 6-36, 6-37
pixel values
defined 4-11
for direct devices 4-15 to 4-17
for indexed devices 4-13 to 4-14
as RGB colors 4-13 to 4-17
PixMap32Bit function 6-39
PixMap data type 4-46 to 4-48. *See also* pixel maps
PixMap records
copying images between 3-112 to 3-122
creating 4-85 to 4-86
disposing of 4-87

PixMap records *(continued)*
low-level routine for copying images between 3-136
obtaining, for offscreen graphics worlds 6-31 to 6-32
pixel images in 4-10 to 4-12
setting 4-86 to 4-87
PixPatChanged procedure 4-98 to 4-99
PixPat data type 4-58 to 4-60. *See also* pixel patterns
PixPatHandle data type 4-58
pixPurge flag 6-13, 6-14, 6-18, 6-19
plus sign cursor 8-8 to 8-9
PnLocHFrac opcode A-6
PnMode opcode A-6, A-18
PnPat opcode A-6, A-18
PnPixPat opcode A-6
PnSize opcode A-6, A-18
Point data type 2-27, A-4. *See also* points
points
adding coordinates of 2-52
assigning coordinates to 2-54
changing between global and local 2-19, 2-51 to 2-52
comparing coordinates of 2-54
coordinates for 2-4 to 2-5
data type for 2-27
defined 1-9 to 1-10
mapping between rectangles 3-106
rectangles around 3-56
relationship to pixels 1-9
routines for managing 2-51 to 2-54, 3-104 to 3-106
subtracting coordinates of 2-53
used for defining rectangles 2-5 to 2-6
whether in rectangles 3-56
whether in regions 3-97
PolyBegin picture comment B-6, B-24, B-28
PolyClose picture comment B-6, B-24
Poly data type A-4
PolyEnd picture comment B-6, B-24
Polygon data type 3-37. *See also* polygons
polygons
closing 3-79
creating 3-78 to 3-79
data type for 3-37
defined 1-15
defining 3-30
disposing of 3-80 to 3-81
drawing 3-81 to 3-85
erasing 3-84
filling
with bit patterns 3-83 to 3-84
with pixel patterns 4-76 to 4-77
framing 3-81 to 3-82
inverting 3-85
low-level routine for drawing 3-135
mapping and scaling 3-108
moving 3-80
painting 3-82 to 3-83
routines for managing 3-78 to 3-85, 3-108
smoothed, on PostScript printers B-23 to B-29
PolyIgnore picture comment B-6, B-24, B-27 to B-28
PolySmooth picture comment B-6, B-24 to B-28
PortChanged procedure 4-99 to 4-100
port rectangles
in basic graphics ports 2-32
changing positions of 2-46 to 2-47
changing sizes of 2-46
changing window origins of 2-23 to 2-26, 2-45 to 2-46
in color graphics ports 4-51
defined 1-7
in graphics ports 2-11
scrolling pixels in 2-20 to 2-26, 2-43 to 2-44
PortSize procedure 2-46
PostScriptBegin picture comment B-8 to B-9, B-31, B-34
PostScriptEnd picture comment B-9, B-31, B-35
PostScriptFile picture comment B-6, B-41
PostScriptHandle picture comment B-6, B-38 to B-39
PostScript language, use in printing B-3 to B-44
PostScript LaserWriter printers 9-76, B-7
PostScript printer drivers 9-9
'ppat' resource type 4-24 to 4-25, 4-103
PrCloseDoc procedure 9-21, 9-22, 9-68
PrClosePage procedure 9-22, 9-70
PrClose procedure 9-22, 9-37, 9-58
PrCtlCall procedure 9-81 to 9-84
PrDlgMain function 9-37, 9-63 to 9-64
PrDrvrClose procedure 9-80
PrDrvrDCE function 9-80 to 9-81
PrDrvrOpen procedure 9-79
PrDrvrVers function 9-79
PrError function 9-18, 9-21, 9-41 to 9-42, 9-75 to 9-77
PrGeneral procedure 9-28 to 9-35, 9-42, 9-72 to 9-74
Print command (File menu) 9-5 to 9-6, 9-7 to 9-8
PrintDefault procedure 9-37, 9-59
print dialog boxes. *See also* job dialog boxes; print status dialog boxes; style dialog boxes
altering 9-35 to 9-38, 9-63 to 9-65, 9-86
data structure for 9-50 to 9-51
displaying 9-61 to 9-64
for multiple documents 9-26, 9-66
print dialog box record. *See* TPrDlg data type
printer drivers
closing 9-58, 9-80
defined 9-3
determining versions of 9-79
device control entry for 9-80 to 9-81
dialog boxes for 9-5 to 9-8, 9-13 to 9-14
line layout capabilities of B-11 to B-17
opening 9-57, 9-79
picture comments supported by B-7
PostScript 9-9

QuickDraw 9-8 to 9-9
resolutions for 9-11, 9-30 to 9-32
printer resource files 9-3
PrintErr global variable 9-78
printers
current, device numbers of 9-48
current, feed types of 9-48
ImageWriter LQ B-7
information in TprInfo records for 9-46
LaserWriter 9-7 to 9-8, 9-76, B-7
LaserWriter SC B-7
Personal LaserWriter LS B-7
PostScript LaserWriter 9-76, B-7
StyleWriter 9-6 to 9-8, B-7
print information record. *See* TPrInfo data type
printing
area for 9-10 to 9-11
canceling 9-14, 9-38 to 9-41, 9-85
deferred 9-24, 9-71 to 9-72
determining number of copies 9-19
determining number of pages 9-19, 9-23
dialog boxes for 9-5 to 9-8, 9-13 to 9-15, 9-50 to 9-51, 9-61 to 9-66
documents 9-18 to 9-26, 9-66 to 9-72
draft-quality 9-24, 9-55
enhanced draft-quality 9-33 to 9-35, 9-55, 9-73
error handling for 9-73, 9-75 to 9-78
from the Finder 9-25 to 9-26, 9-66
graphics ports for. *See* printing graphics ports
landscape, disabled 9-34
multiple documents 9-25 to 9-26, 9-66
with non-QuickDraw features B-3 to B-44
optimizing 9-72 to 9-74
picture comments for B-3 to B-44
resolutions for 9-30 to 9-32, 9-53 to 9-55
status 9-13 to 9-15, 9-49
user interface guidelines for 9-5 to 9-8, 9-13 to 9-15
whether landscape 9-32 to 9-33, 9-56, 9-73
printing graphics ports
closing 9-68
creating 9-19, 9-67
data type for 9-51 to 9-52
defined 9-3 to 9-5
drawing into 9-19 to 9-24, 9-69 to 9-70
opening 9-19, 9-67
printing loops 9-18 to 9-25
Printing Manager 1-26 to 1-28, 9-3 to 9-105
application-defined routines for 9-84 to 9-86
data structures in 9-44 to 9-56
and Dialog Manager 9-5 to 9-8, 9-35 to 9-38
initializing 9-15, 9-57
low-level routines in 9-78 to 9-84
and QuickDraw 9-3 to 9-5
routines in 9-57 to 9-84
testing for availability 9-15
user interface guidelines for 9-5 to 9-8, 9-13 to 9-15
printing status information. *See* TPrStatus data type
printing style record. *See* TPrStl data type
print job record. *See* TPrJob data type
print record. *See* TPrint records
print status dialog boxes 9-13 to 9-15, 9-38 to 9-41
PrJobDialog function 9-20, 9-62 to 9-63
PrJobInit function 9-37, 9-65
PrJobMerge procedure 9-26, 9-66
PrOpenDoc function 9-21, 9-67
PrOpenPage procedure 9-21, 9-69 to 9-70, B-4
PrOpen procedure 9-20, 9-57
PrPicFile procedure 9-21, 9-71 to 9-72
PrSetError procedure 9-78
PrStlDialog function 9-61 to 9-62
PrStlInit function 9-64
PrValidate function 9-18, 9-20, 9-60
PSBeginNoSave picture comment B-6, B-41
Pt2Rect procedure 3-56
PtInRect function 3-56
PtInRgn function 3-97, 8-11
PtToAngle procedure 3-57

Q

QDColor global variable 4-71
QDDone function 3-125 to 3-126
QDError function 3-28, 3-30, 3-34, 4-94 to 4-95, 7-20
QDProcs data type 3-39 to 3-40
QDProcs record B-4
QuickDraw 1-3 to 1-29. *See also* basic QuickDraw; Color QuickDraw; global coordinate systems; local coordinate systems; shapes
compatibility between versions 1-4
customizations of 3-35 to 3-36, 3-129, 4-96 to 4-97
and Dialog Manager 4-6
drawing with 1-10 to 1-17
historical development 1-4
initializing 2-36 to 2-37
low-level drawing routines 3-129 to 3-139
mathematical foundations of 2-4 to 2-7
multiple graphics device support in 1-21 to 1-23
picture comments supported by printer drivers for B-7
printer drivers 9-8 to 9-9
and Printing Manager 9-3 to 9-5
printing with. *See* Printing Manager
text 1-3
versions of 1-4
and the Window Manager 1-7 to 1-8

R

ramInit flag 5-17, 5-23, 5-31, 5-36
randSeed global variable 2-36
reallocPix flag 6-14, 6-15, 6-25
RecordPictInfo function 7-56 to 7-57
RecordPixMapInfo function 7-57 to 7-58
rectangles. *See also* boundary rectangles; bounding rectangles; port rectangles
  coordinates for 2-5 to 2-6
  creating 3-53
  data type for 2-27 to 2-28
  defined 1-12 to 1-13
  defining 3-22 to 3-23, 3-24
  drawing 3-22 to 3-24, 3-58 to 3-62
  emptiness of 3-58
  equality of 3-58
  erasing 3-61 to 3-62
  expanding 3-54
  filling
    with bit patterns 3-23 to 3-24, 3-60 to 3-61
    with pixel patterns 4-74
  framing 3-22 to 3-23, 3-59
  intersections of 3-55
  inverting 3-62
  low-level routine for drawing 3-132
  mapping and scaling 3-106 to 3-107
  moving 3-53 to 3-54
  painting 3-23 to 3-24, 3-60
  pixels in 3-56
  and regions 3-91 to 3-92, 3-98
  routines for managing 3-52 to 3-62, 3-104 to 3-108
  scaling factors for 3-104 to 3-105
  shrinking 3-54
  smallest around two points 3-56
  unions of 3-55
  used to define other shapes 3-11
Rect data type 2-27 to 2-28, A-4. *See also* rectangles
RectInRgn function 3-98
RectRgn procedure 3-92, 8-11
Region data type 2-28 to 2-29. *See also* regions
regions
  arrow 8-9 to 8-12
  copying 3-90 to 3-91
  creating 3-87 to 3-89
  data type for 2-28 to 2-29
  defined 1-16
  defining 3-27 to 3-30
  disposing of 3-90
  drawing 3-100 to 3-104
  emptiness of 3-91, 3-99
  equality of 3-98
  erasing 3-102 to 3-103
  expanding 3-93 to 3-94
  filling
    with bit patterns 3-102
    with pixel patterns 4-77
  framing 3-100 to 3-101
  I-beam 8-9 to 8-12
  intersections of 3-94 to 3-95, 3-96 to 3-97
  inverting 3-103 to 3-104
  low-level routine for drawing 3-135 to 3-136
  mapping and scaling 3-107
  mouse 8-9 to 8-12
  moving 3-93
  painting 3-101
  pixels in 3-97
  and rectangles 3-91 to 3-92, 3-98
  routines for managing 3-85 to 3-104, 3-107
  shrinking 3-93 to 3-94
  subtracting 3-96
  unions of 3-95, 3-96 to 3-97
resolutions
  discrete 9-11
  for pictures 7-11, 7-19
  for printers 9-11, 9-30 to 9-32, 9-46, 9-53 to 9-55, 9-73
  for screens 5-32
  variable 9-11
resource forks 7-7
ResourcePS picture comment B-6, B-41
resources
  animated cursor 8-13, 8-14, 8-36 to 8-37
  color cursor 8-34 to 8-36
  color icon 4-105 to 4-106
  color-picking method 7-68
  color table 4-104 to 4-105
  cursor 8-13 to 8-14, 8-33 to 8-34
  pattern 3-140
  pattern list 3-141
  picture 7-7, 7-20, 7-46, 7-67 to 7-68
  pixel pattern 4-24 to 4-25, 4-103
  screen 5-37
resource types
  'acur' 8-13, 8-14, 8-36 to 8-37
  'cicn' 4-105 to 4-106
  'clut' 4-104 to 4-105
  'cmpt' 7-68
  'crsr' 8-34 to 8-36
  'CURS' 8-13 to 8-14, 8-33 to 8-34
  'PAT ' 3-140
  'PAT#' 3-141
  'PICT' 7-7, 7-20, 7-46, 7-67 to 7-68
  'ppat' 4-24 to 4-25, 4-103
  'scrn' 5-37
RetrievePictInfo function 7-58 to 7-59
RGBBackColor procedure 4-72 to 4-73
RGBBkCol opcode A-6
RGBColorArray data type 7-64
RGBColor data type 4-55. *See also* RGB colors

RGBColor records 1-19, 4-13 to 4-17
RGB colors 1-19
  data type for 4-55
  defined 4-4 to 4-5
  as pixel values 4-13 to 4-17
RGBFgCol opcode A-6
RGBForeColor procedure 4-22, 4-70 to 4-71
Rgn data type A-4
RotateBegin picture comment B-6, B-9, B-29 to B-32
RotateCenter picture comment B-6, B-9, B-32
RotateCursor procedure 8-15, 8-32
RotateEnd picture comment B-6, B-9, B-29, B-32
rounded rectangles
  defined 1-14
  drawing 3-63 to 3-68
  erasing 3-66 to 3-67
  filling
    with bit patterns 3-65 to 3-66
    with pixel patterns 4-74 to 4-75
  framing 3-64
  inverting 3-67 to 3-68
  low-level routine for drawing 3-133
  painting 3-64 to 3-65
RowBytes data type A-4
ruled lines, printing B-33 to B-37

S

sample routines
  DashDemo B-34
  DoControlClick 2-19
  DoGraphicsScroll 2-22
  DoInit 8-6
  DoIsLandscapeModeSet 9-33
  DoNew 2-17, 4-20
  DoPostScriptLine B-39
  DoPrintDialog 9-37
  DoSavePICTAsCmd 7-21
  DoUpdate 5-8
  DoZoomWindow 5-10 to 5-12
  DrawInPort 2-18
  HiliteDemonstration 4-43
  MyAdjustCursor 8-10
  MyAdjustDestRect 7-18
  MyCopyBlackAndRedMasks 6-10
  MyCreateAndDrawPict 7-11, A-22
  MyDefineVertices B-26
  MyDoPrintIdle 9-40
  MyDrawArcAndPaintWedge 3-26
  MyDrawDumbbell 3-28
  MyDrawFilePicture 7-13
  MyDrawLines 3-18
  MyDrawOvals 3-25
  MyDrawRects 3-23
  MyDrawResPICT 7-20
  MyDrawTriangle 3-30
  MyDrawXString B-21
  MyFileGetPic 7-16
  MyFilePutPic 7-23
  MyFillClipRegion 3-29
  MyFlushGrafPortState B-10
  MyFlushPostScriptState B-11
  MyGetPICTProfileCount 7-25
  MyGetPrintRecordForThisDoc 9-17
  MyIsColorPort 7-16
  MyLineWidthDemo B-37
  MyPaintAndFillColorRects 4-22
  MyPaintAndFillRects 3-24
  MyPaintPixelPatternRects 4-25
  MyPaintRectsThruGWorld 6-5
  MyPastePict 7-17
  MyPolygonDemo B-27
  MyPrDialogAppend 9-37
  MyPrintLoop 9-20
  MyRepatternPens 3-21
  MyReplaceGetPic 7-15
  MyReplacePutPic 7-22
  MyResizePens 3-20
  MyRotateCursor 8-15
  MySetHiliteMode 4-42
  MySetNewLineWidth B-37
  MyShrinkImages 3-33
  MySpinCursor 8-15
  MyStringReconDemo B-17
  MyTrivialDrawingProc 5-9
ScalePt procedure 3-104 to 3-105
scrap
  defined 7-7
  pictures in 7-7 to 7-8, 7-17, 7-22
screenActive flag 5-17, 5-23, 5-31, 5-36
screenBits global variable 2-36
screenDevice flag 5-17, 5-23, 5-31, 5-36
screen resources 5-37
ScreenRes procedure 5-32
screens
  determining characteristics of 5-29 to 5-32
  with greatest pixel depth 5-27 to 5-28
  optimizing images for 5-8 to 5-13, 5-29 to 5-30, 5-35 to 5-37
  resolution of 5-32
ScrHRes global variable 5-32
'scrn' resource type 5-37
scrolling pixels 2-20 to 2-26, 2-43 to 2-44
ScrollRect procedure 2-21 to 2-23, 2-43 to 2-44
ScrVRes global variable 5-32
SectRect function 3-55, 5-11
SectRgn procedure 3-94 to 3-95, 8-11
SeedCFill procedure 4-82 to 4-83

SeedFill procedure 3-109 to 3-110
SetCCursor procedure 8-26 to 8-27
SetClip procedure 2-48, 3-29
SetCPixel procedure 4-73
SetCursor procedure 8-11, 8-25
SetDepth function 5-13, 5-34 to 5-35
SetDeviceAttribute procedure 5-22 to 5-23
SetEmptyRgn procedure 3-91
SetFractEnable procedure B-15
SetGDevice procedure 5-24
SetGrayLevel picture comment B-40
SetGWorld procedure 6-6, 6-29
SetLineWidth picture comment B-6, B-35 to B-37
SetOrigin procedure 2-45 to 2-46, 8-11
SetPenState procedure 3-43 to 3-44
SetPixelsState procedure 6-37 to 6-38
SetPortBits procedure 2-50
SetPortPix procedure 4-86 to 4-87
SetPort procedure 2-18, 2-42
SetPt procedure 2-54
SetRect procedure 3-23, 3-25, 3-53, 5-11
SetRectRgn procedure 3-91 to 3-92
setRslOp opcode 9-30 to 9-32, 9-52, 9-54 to 9-55
SetStdCProcs procedure 4-96 to 4-97, 7-15, 7-23
SetStdProcs procedure 3-130
SetWindowPic procedure 7-13, 7-20
shapes. *See also* arcs; lines; ovals; pictures; polygons; rectangles; regions; rounded rectangles; wedges
  calculations and manipulations 3-31 to 3-32
  creating 1-10 to 1-17
  defined 1-10 to 1-17
  defining 3-11 to 3-12
  drawing, erasing, and inverting 3-12 to 3-13
  erasing 1-17
  filling 1-17, 3-108 to 3-112
  framing 1-17
  painting 1-17
ShieldCursor procedure 8-29
ShortComment opcode A-12, A-21
ShortLineFrom opcode A-7, A-19
ShortLine opcode A-7, A-19
Show_Cursor procedure 8-30 to 8-31
ShowCursor procedure 8-30
ShowPen procedure 3-42
singleDevices flag 5-30
source modes 3-8 to 3-11, 4-32 to 4-37
SpExtra opcode A-6, A-18
SpinCursor procedure 8-15, 8-32 to 8-33
spool files 9-8, 9-9, 9-25
srcBic source mode 3-9 to 3-10, 3-114, 3-115, 4-33, 4-34, 4-41
srcCopy source mode 3-9 to 3-10, 3-114, 3-115, 4-33, 4-41
srcOr source mode 3-9 to 3-10, 3-114 to 3-115, 4-33 to 4-34, 4-41
srcXor source mode 3-9 to 3-10, 3-114, 3-115, 4-33, 4-41
StandardGetFile procedure 7-14
standard state of a window 5-10
startup screen 1-23
status, of printing 9-13 to 9-15, 9-38 to 9-41, 9-49
StdArc procedure 3-134
StdBits procedure 3-136
StdComment procedure 3-137, B-4
StdGetPic procedure 3-138 to 3-139
StdLine procedure 3-132, B-24, B-27
StdOval procedure 3-133 to 3-134
StdPoly procedure 3-135
StdPutPic procedure 3-139, 7-14
StdRect procedure 3-132
StdRgn procedure 3-135 to 3-136
StdRRect procedure 3-133
StdText procedure 3-131
StdTxtMeas function 3-138
stretchPix flag 6-14, 6-15, 6-24, 6-25
StringBegin picture comment B-5, B-17
StringEnd picture comment B-5, B-17
style dialog boxes
  altering 9-35 to 9-38, 9-63 to 9-64, 9-86
  defined 9-6
  displaying 9-61 to 9-62
  for LaserWriter printers 9-7
  for StyleWriter printers 9-6 to 9-7
StyleWriter printers 9-6 to 9-8, B-7
subOver arithmetic transfer mode 4-39, 4-40
subPin arithmetic transfer mode 4-39, 4-40, 4-78
SubPt procedure 2-53
System 7 1-4

T

TCenterRec data type B-20 to B-21, B-29
TDashedLineRec data type B-33
TDftBitsBlk data type 9-33 to 9-35, 9-55
TestDeviceAttribute function 5-11, 5-31 to 5-32
text. *See also* text strings
  in basic graphics ports 2-33 to 2-34
  in color graphics ports 4-53
  in graphics ports 2-13
  low-level routine for drawing 3-131
  low-level routine for measuring width 3-138
TextBegin picture comment B-5, B-17 to B-20, B-21
TextCenter picture comment B-5, B-17 to B-18, B-19 to B-21
TextEnd picture comment B-5, B-17 to B-18, B-22
TextIsPostScript picture comment B-6, B-41

text streaming 9-82
text strings
  delimiting with picture comments B-16 to B-17
  rotating with picture comments B-17 to B-22
TFeed data type 9-48
TGetRotnBlk data type 9-32 to 9-33, 9-56
TGetRslBlk data type 9-30 to 9-31, 9-53 to 9-54
TGnlData data type 9-52 to 9-53
TheGDevice global variable 5-4
thePat opcode A-18
thePort global variable 2-36
32-bit Color QuickDraw. *See* Color QuickDraw
TLineWidth data type B-35
TopMapHdl global variable 9-39
TPolyVerbRec data type B-25 to B-26
TPrDlg data type 9-50 to 9-51
TPrInfo data type 9-46
TPrint data type 9-38 to 9-39, 9-44 to 9-46
TPrint records
  creating 9-17
  defined 9-11 to 9-13
  initializing 9-59
  saving and reading 9-17 to 9-18
  validating 9-60
TPrJob data type 9-38 to 9-39, 9-47 to 9-48
TPrPort data type 9-51 to 9-52
TPrPort records
  closing 9-68
  creating 9-19, 9-67
  drawing into 9-24, 9-69 to 9-70
  opening 9-19, 9-67
TPrStatus data type 9-49
TPrStl data type 9-48
transfer modes. *See* arithmetic transfer modes; Boolean transfer modes; pattern modes; source modes
transparent mode 4-39, 4-40
TRotationRec data type B-30
TRslRec data type 9-54
TRslRg data type 9-53
TSetRslBlk data type 9-31, 9-54 to 9-55
TTxtPicRecord data type B-19 to B-20
TxFace opcode A-5, A-18
TxFont opcode A-5, A-18
TxMode opcode A-5, A-18
TxRatio opcode A-6, A-19
TxSize opcode A-6, A-18

U

UnionRect procedure 3-55
UnionRgn procedure 3-95
UnlockPixels procedure 6-6, 6-33 to 6-34
UpdateGWorld function 6-9, 6-23 to 6-26
user interface guidelines
  for animated cursors 8-5, 8-13, 8-15
  for color cursors 8-5
  for cursors 8-4 to 8-5
  for highlighting 4-44
  for Printing Manager 9-13 to 9-15
  for style and job dialog boxes 9-5 to 9-8
user state of a window 5-9
useTempMem flag 6-13, 6-14, 6-18, 6-20

V

variable resolution 9-11, 9-30 to 9-32
version 1 format 7-5 to 7-6, A-3 to A-4, A-5, A-18 to A-21, A-25 to A-26
version 2 format 7-5 to 7-6, 7-39, A-3, A-5 to A-16, A-24 to A-25
Version opcode A-6, A-13
video devices 1-19 to 1-20, 1-22 to 1-25, 5-3 to 5-37
visible regions 2-11
  in basic graphics ports 2-32
  in color graphics ports 4-51

W

wedges. *See also* arcs
  defined 1-14
  drawing 3-26, 3-71 to 3-77
  erasing 3-76
  filling
    with bit patterns 3-75
    with pixel patterns 4-76
  inverting 3-77
  low-level routine for drawing 3-134
  painting 3-73 to 3-74
white global variable 2-36, 3-7
Window Manager
  and pictures 7-13
  and QuickDraw 1-7 to 1-8
window origins
  changing 2-23 to 2-26, 2-45 to 2-46
  defined 2-20
windows
  as graphics ports 1-7 to 1-8
  scrolling through 2-20 to 2-26, 2-43 to 2-44
  standard state 5-10
  updating 2-24
  user state 5-9
  zooming 5-9 to 5-12
wristwatch cursor 8-8 to 8-9

X, Y

`XorRgn` procedure 3-96 to 3-97

Z

`0..255` data type A-4
zooming windows 5-9 to 5-12
`ZoomWindow` procedure 5-10, 5-12