




Mac OS For QuickTime Programmers



Apple Computer, Inc.
Technical Publications
April, 1998

 Apple Computer, Inc.
© 1998 Apple Computer, Inc.
All rights reserved.

No part of this publication or the software described in it 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., except in the normal use of the software or to make a backup copy of the software or documentation. The same proprietary and copyright notices must be affixed to any permitted copies as were affixed to the original. This exception does not allow copies to be made for others, whether or not sold, but all of the material purchased (with all backup copies) may be sold, given, or loaned to another person. Under the law, copying includes translating into another language or format. You may use the software on any computer owned by you, but extra copies cannot be made for this purpose.

The Apple logo is a trademark of Apple Computer, Inc. Use of the “keyboard” Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

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. 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.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, Mac, Macintosh, QuickDraw, and QuickTime are trademarks of Apple Computer, Inc., registered in the United States and other countries.

The QuickTime logo is a trademark of Apple Computer, Inc. Adobe, Acrobat, Photoshop, and PostScript are trademarks of Adobe Systems Incorporated or its subsidiaries and may be registered in certain jurisdictions.

Helvetica and Palatino are registered trademarks of Linotype-Hell AG and/or its subsidiaries.

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

Simultaneously published in the United States and Canada.

Printed in the United States of America.

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 DISTRIBUTION 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 DISTRIBUTED “AS IS,” AND YOU 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	13
Preface	About This Book	19
	Sources of Material	19
	Conventions Used in This Book	19
	Special Fonts	20
	Types of Notes	20
Chapter 1	Overview and Summary	21
	Component Management	22
	Finding Components	22
	Opening and Closing Components	23
	Getting Information About a Component	23
	Retrieving Component Errors	23
	Component Functions	23
	Registering Components	24
	Dispatching to Component Routines	24
	Managing Component Connections	24
	Miscellaneous Component Functions	25
	Component Manager Data Structures	25
	Component Resources	26
	Component Manager Constants	26
	Mac OS File System	26
	Memory Management	27
	Mac OS Events	28
	Window Management	28
	QuickDraw	28
	Drawing Environment	29
	Graphics Devices	29
	QuickDraw Operations Offscreen	29
	Pictures	30
Chapter 2	Component Manager	31
	Introduction to Components	31
	About the Component Manager	33
	Using the Component Manager	34
	Opening Connections to Components	35

Opening a Connection to a Default Component	35
Finding a Specific Component	36
Opening a Connection to a Specific Component	38
Getting Information About a Component	38
Using a Component	39
Closing a Connection to a Component	40
Creating Components	41
The Structure of a Component	41
Handling Requests for Service	46
Responding to the Open Request	47
Responding to the Close Request	49
Responding to the Can Do Request	50
Responding to the Version Request	50
Responding to the Register Request	51
Responding to the Unregister Request	52
Responding to the Target Request	52
Responding to Component-Specific Requests	54
Reporting an Error Code	56
Defining a Component's Interfaces	56
Managing Components	58
Registering a Component	58
Creating a Component Resource	60
Establishing and Managing Connections	62
Component Manager Reference	65
Data Structures for Applications	65
The Component Description Record	65
Component Identifiers and Component Instances	68
Routines for Applications	69
Finding Components	70
Opening and Closing Components	72
Getting Information About Components	75
Retrieving Component Errors	79
Data Structures for Components	80
The Component Description Record	80
The Component Parameters Record	82
Routines for Components	84
Registering Components	85
Dispatching to Component Routines	91
Managing Component Connections	93
Setting Component Errors	97
Working With Component Reference Constants	98
Accessing a Component's Resource File	99
Calling Other Components	101
Capturing Components	102
Targeting a Component Instance	104
Changing the Default Search Order	105

Application-Defined Routine	106
Resources	107
The Component Resource	107
Constants	114
Result Codes	115

Chapter 3	Introduction to File Management	117
-----------	---------------------------------	-----

About Files	118
File Forks	118
File Size	120
File Access Characteristics	122
The Hierarchical File System	123
Identifying Files and Directories	126
Using Files	126
Testing for File Management Routines	128
Defining a Document Record	129
Creating a New File	130
Opening a File	132
Reading File Data	136
Writing File Data	137
Saving a File	140
Reverting to a Saved File	144
Closing a File	146
Opening Files at Application Startup Time	148
Using a Preferences File	150
Adjusting the File Menu	151
File Management Reference	152
Data Structures	153
File System Specification Record	153
Standard File Reply Records	153
Application Files Records	155
File Specification Routines	156
File Access Routines	157
Reading, Writing, and Closing Files	158
Manipulating the File Mark	160
Manipulating the End-of-File	162
File and Directory Manipulation Routines	163
Opening, Creating, and Deleting Files	163
Exchanging the Data in Two Files	167
Creating File System Specifications	168
Volume Access Routines	169
Updating Volumes	169
Obtaining Volume Information	170
Application Launch File Routines	171
Result Codes	174

Chapter 4	Standard File Package	177
	About the Standard File Package	178
	Standard User Interfaces	179
	Opening Files	179
	Saving Files	180
	Keyboard Equivalents	182
	Customized User Interfaces	182
	Saving Files	182
	Opening Files	184
	Selecting Volumes and Directories	184
	User Interface Guidelines	187
	Using the Standard File Package	188
	Presenting the Standard User Interface	189
	Customizing the User Interface	191
	Customizing Dialog Boxes	192
	Writing a File Filter Function	195
	Writing a Dialog Hook Function	196
	Writing a Modal-Dialog Filter Function	203
	Writing an Activation Procedure	205
	Setting the Current Directory	206
	Selecting a Directory	209
	Selecting a Volume	213
	Using the Original Procedures	215
	Standard File Package Reference	216
	Data Structures	216
	Enhanced Standard File Reply Record	217
	Original Standard File Reply Record	218
	Standard File Package Routines	219
	Saving Files	219
	Opening Files	224
	Application-Defined Routines	230
	File Filter Functions	230
	Dialog Hook Functions	231
	Modal-Dialog Filter Functions	232
	Activation Procedures	234
Chapter 5	Introduction to Memory Management	235
	About Memory	236
	Organization of Memory by the Operating System	236
	The System Heap	238
	The System Global Variables	238
	Organization of Memory in an Application Partition	239
	The Application Stack	240
	The Application Heap	241

The Application Global Variables and A5 World	244
Temporary Memory	245
Virtual Memory	247
Addressing Modes	247
Heap Management	248
Relocatable and Nonrelocatable Blocks	248
Properties of Relocatable Blocks	252
Locking and Unlocking Relocatable Blocks	252
Purging and Reallocating Relocatable Blocks	253
Memory Reservation	254
Heap Purging and Compaction	255
Heap Fragmentation	256
Deallocating Nonrelocatable Blocks	257
Reserving Memory	257
Locking Relocatable Blocks	258
Allocating Nonrelocatable Blocks	259
Summary of Preventing Fragmentation	260
Dangling Pointers	261
Compiler Dereferencing	261
Loading Code Segments	263
Callback Routines	264
Invalid Handles	265
Disposed Handles	265
Empty Handles	266
Fake Handles	267
Low-Memory Conditions	268
Memory Cushions	268
Memory Reserves	269
Grow-Zone Functions	270
Using Memory	270
Setting Up the Application Heap	270
Changing the Size of the Stack	271
Expanding the Heap	272
Allocating Master Pointer Blocks	273
Determining the Amount of Free Memory	274
Allocating Blocks of Memory	276
Maintaining a Memory Reserve	278
Defining a Grow-Zone Function	280
Memory Management Reference	282
Memory Management Routines	282
Setting Up the Application Heap	282
Allocating and Releasing Relocatable Blocks of Memory	286
Allocating and Releasing Nonrelocatable Blocks of Memory	290
Setting the Properties of Relocatable Blocks	292
Managing Relocatable Blocks	299
Manipulating Blocks of Memory	305
Assessing Memory Conditions	307

Grow-Zone Operations	309
Setting and Restoring the A5 Register	310
Application-Defined Routines	312
Grow-Zone Functions	312
Result Codes	313

Chapter 6 **Memory Manager** 315

About the Memory Manager	316
Temporary Memory	316
Multiple Heap Zones	317
The System Global Variables	319
Using the Memory Manager	320
Reading and Writing System Global Variables	320
Extending an Application's Memory	322
Allocating Temporary Memory	323
Determining the Features of Temporary Memory	323
Using the System Heap	324
Allocating Memory at Startup Time	326
Creating Heap Zones	327
Installing a Purge-Warning Procedure	329
Organization of Memory	331
Heap Zones	332
Block Headers	334
Memory Manager Reference	337
Data Types	337
Memory Manager Routines	338
Setting Up the Application Heap	339
Allocating and Releasing Relocatable Blocks of Memory	341
Allocating and Releasing Nonrelocatable Blocks of Memory	348
Changing the Sizes of Relocatable and Nonrelocatable Blocks	351
Setting the Properties of Relocatable Blocks	356
Managing Relocatable Blocks	364
Manipulating Blocks of Memory	372
Assessing Memory Conditions	379
Freeing Memory	384
Grow-Zone Operations	389
Allocating Temporary Memory	390
Accessing Heap Zones	393
Manipulating Heap Zones	396
Application-Defined Routines	402
Grow-Zone Functions	402
Purge-Warning Procedures	403

Event Manager Constants and Types	407
Event Kinds	407
Event Masks	409
Event Modifier Flags	411
Posting Options	412
The Key Map Type	413
The Event Record	414
The Target ID Structure	417
The High-Level Event Message Structure	418
The Event Queue	419
Filter Function Pointer and Macro	420
Event Manager Functions	421
Receiving Events	422
Sending Events	442
Converting Process Serial Numbers and Port Names	449
Reading the Mouse	452
Reading the Keyboard	455
Getting Timing Information	457
Accessors for Low-Memory Globals	462
Application-Defined Function	467
Filter Function for Searching the High-Level Event Queue	467
Event Manager Resource	468
The Size Resource	468

Window Manager Constants and Data Types	475
Window Definition IDs	476
FindWindow Result Codes	479
Window Kinds	481
Part Identifiers for ColorSpec Records	482
The Color Window Record	483
The Window Record	488
The Window State Data Record	489
The Window Color Table Record	490
The Auxiliary Window Record	492
The Window List	494
Window Definition Function Type and Macros	494
Window Definition Function Enumerators	495
Window Manager Functions	497
Initializing the Window Manager	497
Creating Windows	498
Naming Windows	509
Displaying Windows	511

Retrieving Window Information	517
Moving Windows	518
Resizing Windows	526
Zooming Windows	528
Closing and Deallocating Windows	531
Maintaining the Update Region	534
Setting and Retrieving Other Window Characteristics	537
Manipulating the Desktop	540
Manipulating Window Color Information	544
Low-Level Functions	546
Accessors for Low-Memory Globals	551
Application-Defined Function	562
Window Definition Function	563
Resources	567
The Window Resource	568
The Window Definition Function Resource	571
The Window Color Table Resource	571

Chapter 9

Color QuickDraw 573

About Color QuickDraw	574
RGB Colors	574
The Color Drawing Environment: Color Graphics Ports	575
Pixel Maps	579
Pixel Patterns	582
Color QuickDraw's Translation of RGB Colors to Pixel Values	583
Colors on Grayscale Screens	587
Using Color QuickDraw	588
Initializing Color QuickDraw	589
Creating Color Graphics Ports	590
Drawing With Different Foreground Colors	591
Drawing With Pixel Patterns	593
Copying Pixels Between Color Graphics Ports	596
Boolean Transfer Modes With Color Pixels	602
Dithering	607
Arithmetic Transfer Modes	608
Highlighting	611
Color QuickDraw Reference	614
Data Structures	615
Color QuickDraw Routines	633
Opening and Closing Color Graphics Ports	633
Managing a Color Graphics Pen	637
Changing the Background Pixel Pattern	638
Drawing With Color QuickDraw Colors	640
Determining Current Colors and Best Intermediate Colors	649
Calculating Color Fills	652

Creating, Setting, and Disposing of Pixel Maps	655
Creating and Disposing of Pixel Patterns	657
Creating and Disposing of Color Tables	661
Retrieving Color QuickDraw Result Codes	664
Customizing Color QuickDraw Operations	666
Reporting Data Structure Changes to QuickDraw	667
Application-Defined Routine	671
Resources	672
The Pixel Pattern Resource	673
The Color Table Resource	674
The Color Icon Resource	675
Result Codes	677

Chapter 10 Graphics Devices 679

About Graphics Devices	679
Using Graphics Devices	682
Optimizing Your Images for Different Graphics Devices	684
Zooming Windows on Multiscreen Systems	685
Setting a Device's Pixel Depth	689
Exceptional Cases When Working With Color Devices	689
Graphics Devices Reference	690
Data Structures	691
Routines for Graphics Devices	695
Creating, Setting, and Disposing of GDevice Records	695
Getting the Available Graphics Devices	701
Determining the Characteristics of a Video Device	705
Changing the Pixel Depth for a Video Device	709
Application-Defined Routine	711
Resource	713
The Screen Resource	713

Chapter 11 Offscreen Graphics Worlds 715

About Offscreen Graphics Worlds	716
Using Offscreen Graphics Worlds	716
Creating an Offscreen Graphics World	717
Setting the Graphics Port for an Offscreen Graphics World	720
Drawing Into an Offscreen Graphics World	720
Copying an Offscreen Image Into a Window	721
Updating an Offscreen Graphics World	721
Creating a Mask and a Source Image in Offscreen Graphics Worlds	722
Offscreen Graphics Worlds Reference	724
Data Structures	724

Routines	728
Creating, Altering, and Disposing of Offscreen Graphics Worlds	728
Saving and Restoring Graphics Ports and Offscreen Graphics Worlds	739
Managing an Offscreen Graphics World's Pixel Image	742
Result Codes	752

Chapter 12

Pictures 753

About Pictures	754
Picture Formats	755
Opcodes: Drawing Commands and Picture Comments	756
Color Pictures in Basic Graphics Ports	756
'PICT' Files, 'PICT' Resources, and the 'PICT' Scrap Format	757
The Picture Utilities	758
Using Pictures	758
Creating and Drawing Pictures	760
Opening and Drawing Pictures	763
Drawing a Picture Stored in a 'PICT' File	763
Drawing a Picture Stored in the Scrap	767
Defining a Destination Rectangle	768
Drawing a Picture Stored in a 'PICT' Resource	769
Saving Pictures	770
Gathering Picture Information	773
Pictures Reference	775
Data Structures	776
QuickDraw and Picture Utilities Routines	785
Creating and Disposing of Pictures	785
Drawing Pictures	792
Collecting Picture Information	795
Application-Defined Routines	810
Resources	816
The Picture Resource	816
The Color-Picking Method Resource	817
Result Codes	818

Index 819

Figures, Tables, and Listings

Chapter 1	Overview and Summary	21
Chapter 2	Component Manager	31
	Figure 2-1	The relationship between an application, the Component Manager, and components 34
	Listing 2-1	Finding a component 37
	Listing 2-2	Opening a specific component 38
	Listing 2-3	Getting information about a component 38
	Listing 2-4	Using a drawing component 39
	Table 2-1	Request codes 42
	Listing 2-5	A drawing component for ovals 44
	Listing 2-6	Responding to an open request 48
	Listing 2-7	Responding to a close request 49
	Listing 2-8	Responding to the can do request 50
	Listing 2-9	Responding to the setup request 54
	Listing 2-10	Responding to the draw request 54
	Listing 2-11	Responding to the erase request 55
	Listing 2-12	Responding to the click request 55
	Listing 2-13	Responding to the move to request 56
	Listing 2-14	Registering a component 59
	Listing 2-15	Rez input for a component resource 61
	Figure 2-2	Supporting multiple component connections 62
	Listing 2-16	Delegating a request to another component 64
	Figure 2-3	Interaction between the <code>componentFlags</code> and <code>componentFlagsMask</code> fields 68
	Figure 2-4	Format of a component file 112
	Figure 2-5	Structure of a compiled component (<code>'thng'</code>) resource 113
Chapter 3	Introduction to File Management	117
	Figure 3-1	The two forks of a Macintosh file 119
	Figure 3-2	Logical blocks and allocation blocks 121
	Figure 3-3	Logical end-of-file and physical end-of-file 122
	Figure 3-4	The Macintosh hierarchical file system 124
	Figure 3-5	The disk switch dialog box 125
	Figure 3-6	A typical File menu 126
	Listing 3-1	Handling the File menu commands 127
	Listing 3-2	Testing for the availability of routines that operate on <code>FSSpec</code> records 128
	Listing 3-3	A sample document record 129
	Listing 3-4	Handling the New menu command 130
	Listing 3-5	Creating a new document window 131
	Listing 3-6	Handling the Open menu command 133

Figure 3-7	The default Open dialog box	133
Listing 3-7	Opening a file	134
Listing 3-8	Reading data from a file	136
Listing 3-9	Writing data into a file	138
Listing 3-10	Updating a file safely	139
Listing 3-11	Handling the Save menu command	140
Listing 3-12	Handling the Save As menu command	141
Figure 3-8	The default Save dialog box	142
Figure 3-9	The new folder dialog box	143
Figure 3-10	The name conflict dialog box	143
Listing 3-13	Copying a resource from one resource fork to another	144
Figure 3-11	A Revert to Saved dialog box	144
Listing 3-14	Handling the Revert to Saved menu command	145
Listing 3-15	Handling the Close menu command	146
Listing 3-16	Closing a file	147
Listing 3-17	Opening files at application launch time	149
Listing 3-18	Opening a preferences file	150
Listing 3-19	Adjusting the File menu	151

Chapter 4

Standard File Package 177

Figure 4-1	The default Open dialog box	179
Figure 4-2	The default Save dialog box	180
Figure 4-3	The New Folder dialog box	181
Figure 4-4	The name conflict dialog box	181
Figure 4-5	The Save dialog box customized with radio buttons	183
Figure 4-6	The Save dialog box customized with a pop-up menu	183
Figure 4-7	The Open dialog box customized with a pop-up menu	184
Figure 4-8	The Open dialog box customized to allow selection of a directory	185
Figure 4-9	The Open dialog box when no directory is selected	186
Figure 4-10	The Open dialog box with a long directory name abbreviated	186
Figure 4-11	A volume selection dialog box	187
Listing 4-1	Handling the Open menu command	189
Listing 4-2	Specifying more than four file types	190
Listing 4-3	Presenting a customized Open dialog box	192
Listing 4-4	The definition of the default Open dialog box	193
Listing 4-5	The definition of the default Save dialog box	193
Listing 4-6	The item list for the default Open dialog box	193
Listing 4-7	The item list for the default Save dialog box	194
Listing 4-8	A sample file filter function	196
Listing 4-9	A sample dialog hook function	202
Listing 4-10	A sample modal-dialog filter function	205
Listing 4-11	Determining the current directory	206
Listing 4-12	Determining the current volume	207
Listing 4-13	Setting the current directory	207
Listing 4-14	Setting the current volume	207
Listing 4-15	Setting the current directory	208
Listing 4-16	A file filter function that lists only directories	209

Listing 4-17	Setting a button's title	210
Listing 4-18	Handling user selections in the directory selection dialog box	210
Listing 4-19	Presenting the directory selection dialog box	212
Listing 4-20	A file filter function that lists only volumes	213
Listing 4-21	Handling user selections in the volume selection dialog box	214
Listing 4-22	Presenting the volume selection dialog box	215

Chapter 5

Introduction to Memory Management 235

Figure 5-1	Memory organization with several applications open	237
Figure 5-2	Organization of an application partition	239
Figure 5-3	The application stack	241
Figure 5-4	A fragmented heap	242
Figure 5-5	A compacted heap	243
Figure 5-6	Organization of an application's A5 world	244
Figure 5-7	Using temporary memory allocated from unused RAM	246
Figure 5-8	A pointer to a nonrelocatable block	249
Figure 5-9	A handle to a relocatable block	251
Figure 5-10	Purging and reallocating a relocatable block	254
Figure 5-11	Allocating a nonrelocatable block	255
Figure 5-12	An effectively partitioned heap	258
Listing 5-1	Locking a block to avoid dangling pointers	262
Listing 5-2	Creating a fake handle	267
Listing 5-3	Increasing the amount of space allocated for the stack	272
Listing 5-4	Setting up your application heap and stack	274
Listing 5-5	Determining whether allocating memory would deplete the memory cushion	275
Listing 5-6	Allocating relocatable blocks	276
Listing 5-7	Allocating nonrelocatable blocks	277
Listing 5-8	Allocating a dialog record	277
Listing 5-9	Creating an emergency memory reserve	278
Listing 5-10	Checking the emergency memory reserve	279
Listing 5-11	Determining whether allocating memory would deplete the memory cushion	279
Listing 5-12	Reallocating the emergency memory reserve	280
Listing 5-13	A grow-zone function that releases emergency storage	281

Chapter 6

Memory Manager 315

Listing 6-1	Reading the value of a system global variable	321
Listing 6-2	Changing the value of a system global variable	321
Listing 6-3	Determining whether temporary-memory routines are available	324
Listing 6-4	Calling a procedure by address	326
Listing 6-5	Creating a subzone of the original application heap zone	328
Listing 6-6	A purge-warning procedure	329
Listing 6-7	Installing a purge-warning procedure	330
Listing 6-8	A purge-warning procedure that calls the Resource Manager's procedure	331

Figure 6-1	A block header in a 24-bit zone	335
Figure 6-2	A block header in a 32-bit zone	336

Chapter 7

Event Manager 407

Figure 7-1	Structure of the <code>KeyTranslate</code> function result	457
Listing 7-1	A Rez template for a 'SIZE' resource	470

Chapter 8

Window Manager 475

Figure 7-2	Limiting rectangle used by <code>DragGrayRgn</code>	524
Figure 7-3	Structure of a compiled window ('WIND') resource	568
Figure 7-4	Structure of a compiled window color table ('wctb') resource	572

Chapter 9

Color QuickDraw 573

Figure 8-1	The color graphics port	577
Figure 8-2	The pixel map	580
Figure 8-3	Translating a 48-bit <code>RGBColor</code> record to an 8-bit pixel value on an indexed device	584
Figure 8-4	Translating an 8-bit pixel value on an indexed device to a 48-bit <code>RGBColor</code> record	585
Figure 8-5	Translating a 48-bit <code>RGBColor</code> record to a 32-bit pixel value on a direct device	585
Figure 8-6	Translating a 48-bit <code>RGBColor</code> record to a 16-bit pixel value on a direct device	586
Figure 8-7	Translating a 32-bit pixel value to a 48-bit <code>RGBColor</code> record	586
Figure 8-8	Translating a 16-bit pixel value to a 48-bit <code>RGBColor</code> record	587
Listing 8-1	Using the Window Manager to create a color graphics port	590
Listing 8-2	Changing the foreground color	592
Figure 8-9	Drawing with two different foreground colors (on a grayscale screen)	593
Figure 8-10	Using <code>ResEdit</code> to create a pixel pattern resource	594
Listing 8-3	Rez input for a pixel pattern resource	594
Listing 8-4	Using pixel patterns to paint and fill	595
Figure 8-11	Painting and filling rectangles with pixel patterns	595
Figure 8-12	Copying pixel images with the <code>CopyBits</code> procedure	597
Figure 8-13	Copying pixel images with the <code>CopyMask</code> procedure	599
Figure 8-14	Copying pixel images with the <code>CopyDeepMask</code> procedure	601
Table 8-1	Boolean source modes with colored pixels	603
Listing 8-5	Using <code>CopyBits</code> to produce coloration effects	605
Table 8-2	Arithmetic modes in a 1-bit environment	610
Figure 8-15	Difference between highlighting and inverting	612
Listing 8-6	Setting the highlight bit	612
Listing 8-7	Using highlighting for text	613
Table 8-3	Initial values in the <code>CGrafPort</code> record	634
Table 8-4	The colors defined by the global variable <code>QDColors</code>	641
Table 8-5	The default color tables for grayscale graphics devices	662
Table 8-6	The default color tables for color graphics devices	663

Figure 8-16	Format of a compiled pixel pattern ('ppat') resource	673
Figure 8-17	Format of a compiled color table ('clut') resource	674
Figure 8-18	Format of a compiled color icon ('cicn') resource	676

Chapter 10

Graphics Devices 679

Figure 9-1	The GDevice record	681
Listing 9-1	Using the DeviceLoop procedure	684
Listing 9-2	Drawing into different screens	685
Listing 9-3	Zooming a window	686

Chapter 11

Offscreen Graphics Worlds 715

Listing 10-1	Using a single offscreen graphics world and the CopyBits procedure	718
Listing 10-2	Using two offscreen graphics worlds and the CopyMask procedure	722

Chapter 12

Pictures 753

Figure 11-1	A picture of a party hat	754
Figure 11-2	The Picture record	755
Listing 11-1	Creating and drawing a picture	761
Figure 11-3	A simple picture	762
Listing 11-2	Opening and drawing a picture from disk	763
Listing 11-3	Replacing QuickDraw's standard low-level picture-reading routine	765
Listing 11-4	Determining whether a graphics port is color or basic	766
Listing 11-5	A custom low-level procedure for spooling a picture from disk	766
Listing 11-6	Pasting in a picture from the scrap	767
Listing 11-7	Adjusting the destination rectangle for a picture	768
Listing 11-8	Drawing a picture stored in a resource file	769
Listing 11-9	Saving a picture as a 'PICT' file	770
Listing 11-10	Replacing QuickDraw's standard low-level picture-writing routine	772
Listing 11-11	A custom low-level routine for spooling a picture to disk	773
Listing 11-12	Looking for color profile comments in a picture	774
Table 11-1	Routine selectors for an application-defined color-picking method	810
Figure 11-4	Structure of a compiled picture ('PICT') resource	817

About This Book

Although QuickTime 3 is truly cross-platform—you use one API for compilation to either Windows or the Mac OS—developers for the Windows platform need to understand some facets of Mac OS programming. This book describes the main Mac OS calls and data structures that developers of QuickTime 3 applications may need to use.

Sources of Material

The chapters of this book are reprints of the following chapters from *Inside Macintosh*, with corrections to bring them up to date:

- *Inside Macintosh: More Macintosh Toolbox*
 - Chapter 6, “Component Manager”
- *Inside Macintosh: Files*
 - Chapter 1, “Introduction to File Management”
 - Chapter 3, “Standard File Package”
- *Inside Macintosh: Memory*
 - Chapter 1, “Introduction to Memory Management”
 - Chapter 2, “Memory Manager”
- *Inside Macintosh: Macintosh Toolbox Essentials*
 - Chapter 2, “Event Manager”
 - Chapter 4, “Window Manager”
- *Inside Macintosh: Imaging with QuickDraw*
 - Chapter 4, “Color QuickDraw”
 - Chapter 5, “Graphics Devices”
 - Chapter 6, “Offscreen Graphics Worlds”
 - Chapter 7, “Pictures”

Conventions Used in This Book

This book uses special conventions to present certain types of information. Words that require special treatment appear in specific fonts or font styles. Certain information, such as parameter blocks, appears in special formats so that you can scan it quickly.

Special Fonts

All code listings, reserved words, and the names of actual data structures, constants, fields, 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. ♦

IMPORTANT

A note like this contains information that is essential for an understanding of the main text. ▲

▲ **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. ▲

Overview and Summary

This chapter summarizes the Mac OS functions and data structures that you are most likely to use when writing code for QuickTime 3. They are all supported in current versions of the Mac OS and have also been ported to QuickTime for Windows 95 and Windows NT.

These functions and data structures perform system-level tasks related to QuickTime processes. They are described in full technical detail in the subsequent chapters of this book:

- Chapter 2, “Component Manager,” tells you how to find, access, and create Mac OS components, the basic units from which the QuickTime software is built.
- Chapter 3, “Introduction to File Management,” and Chapter 4, “Standard File Package,” describe the Macintosh file system and tell you how to use the Macintosh standard file package to manage the user interface for naming and identifying files.
- Chapter 5, “Introduction to Memory Management,” and Chapter 6, “Memory Manager,” tell you how to create and dispose of pointers and handles, both within an application’s memory partition and outside the partition.
- Chapter 7, “Event Manager,” describes Macintosh event handling and the event record structure.
- Chapter 8, “Window Manager,” describes Mac OS window records and graphics ports.
- Chapter 9, “Color QuickDraw,” and Chapter 10, “Graphics Devices,” tell you how to use Mac OS color graphics ports and graphics device records.
- Chapter 11, “Offscreen Graphics Worlds,” describes the Mac OS functions and data structures that QuickTime 3 programmers can access to use QuickDraw offscreen.

- Chapter 12, “Pictures,” tells you how to store and draw graphic images in picture records.

Much of the content of this book consists of background information that helps you understand how the Mac OS works. The purpose of this overview and summary chapter is to identify and highlight the particular API elements that you are most likely to use with QuickTime 3.

IMPORTANT

If you are a Windows developer and are not familiar with the Mac OS, first read *QuickTime 3 For Windows Programmers*. PDF and HTML versions of this book are included in the QuickTime 3 SDK. ▲

Component Management

Chapter 2, “Component Manager,” in this book defines some of the original Mac OS Component Manager functions and data structures that a QuickTime application may call when interacting with a component. Newer Component Manager functions are described in Chapter 2, “Component Manager,” of *QuickTime 3 Reference*.

The “Component Manager” road map page, in the online version of the QuickTime 3 documentation, contains links to both the old and newer API elements.

Finding Components

- `FindNextComponent` (page 70) returns the component identifier for the next registered component that meets the selection criteria specified by an application.
- `CountComponents` (page 71) determines the number of registered components that meet the given selection criteria.
- `GetComponentListModSeed` (page 72) lets you determine if the list of registered components has changed.

Opening and Closing Components

- `OpenDefaultComponent` (page 73) lets an application gain access to the services provided by a component that is specified by component type and subtype.
- `OpenComponent` (page 74) lets an application gain access to the services provided by a component that is specified by a component identifier previously obtained from the `FindNextComponent` function.
- `CloseComponent` (page 75) terminates your application's access to the services provided by a component.

Getting Information About a Component

- `GetComponentInfo` (page 76) returns all of the registration information for a component.
- `GetComponentInfo` (page 76) returns a handle to the component's icon suite (if it provides one).
- `GetComponentVersion` (page 78) returns a component's version number.
- `ComponentFunctionImplemented` (page 79) lets you determine whether a component supports a specified request.

Retrieving Component Errors

- `GetComponentInstanceError` (page 79) returns the last error generated by a specific connection to a component.

Component Functions

“Routines for Components” (page 84) defines the Component Manager functions that are used by components. This material will be of interest primarily to developers who are creating components.

“Application-Defined Routine” (page 106) describes how to define a component function and supply the appropriate registration information.

Registering Components

- `RegisterComponent` (page 85) makes a component that is resident in memory available for use by applications or other clients.
- `RegisterComponentResource` (page 87) makes a component that is stored in a resource available for use by applications or other clients.
- `RegisterComponentResourceFile` (page 89) registers all component resources in a given resource file.
- `UnregisterComponent` (page 90) removes a component from the Component Manager's registration list.

Dispatching to Component Routines

- `CallComponentFunction` (page 91) invokes a specified function of your component with the parameters originally provided by the application that called your component.
- `CallComponentFunctionWithStorage` (page 92) invokes a specified function of your component with the parameters originally provided by the application that called your component and provides a handle to the memory associated with the current connection.

Managing Component Connections

- `SetComponentInstanceStorage` (page 94) lets a component pass the Component Manager a handle to the memory that was allocated for the connection to it.
- `GetComponentInstanceStorage` (page 95) lets your component retrieve a handle to the memory associated with a connection.
- `CountComponentInstances` (page 95) lets you determine the number of open connections being managed by a specified component.
- `SetComponentInstanceA5` (page 96) procedure lets your component set the Macintosh A5 register world for a connection.
- `GetComponentInstanceA5` (page 96) retrieves the value of the A5 register for a specified connection.

Miscellaneous Component Functions

The following functions are used for retrieving component errors, working with component reference constants, accessing a component's resource file, calling other components, capturing components, targeting a component instance, and changing the default search order for components.

- `SetComponentInstanceError` (page 97) procedure lets a component pass error information to the Component Manager other than through its return value.
- `SetComponentRefcon` (page 98) procedure sets the reference constant for a component.
- `GetComponentRefcon` (page 99) retrieves the value of the reference constant for your component.
- `OpenComponentResFile` (page 99) allows your component to gain access to its resource file.
- `CloseComponentResFile` (page 100) closes the resource file that your component opened previously with the `OpenComponentResFile` function.
- `DelegateComponentCall` (page 101) provides an efficient mechanism for passing on requests to a specified component.
- `CaptureComponent` (page 103) allows your component to capture another component.
- `UncaptureComponent` (page 104) lets a component uncapture a previously captured component.
- `ComponentSetTarget` (page 104) calls a component's target request routine, which handles the `kComponentTargetSelect` request code.
- `SetDefaultComponent` (page 105) lets a component change the search order for registered components.

Component Manager Data Structures

"Data Structures for Applications" (page 65) defines the data structures your application can use to interact with components.

- The **component description** record `ComponentDescription` identifies the characteristics of a component, including the type of services offered by the component and its manufacturer. It is used by `FindNextComponent`,

`CountComponents`, `GetComponentInfo`, and `RegisterComponent`. See “The Component Description Record” (page 80).

- **Component identifiers and component instances** are two similar ways of specifying components. They are used by most Component Manager routines. See “Component Identifiers and Component Instances” (page 68).
- **The component parameters record** `ComponentParameters` is used to pass parameters, along with information about the current connection, between an application and a component. It is used by the component and by `CallComponentFunction`, `CallComponentFunctionWithStorage`, and `DelegateComponentCall`. See “The Component Parameters Record” (page 83).

Component Resources

“Resources” (page 107) describes the format and content of the Macintosh resources you use when defining a component. If you are developing a component, you should be familiar with this material.

Component Manager Constants

“Constants” (page 114) lists the constants provided by the Component Manager. These constants provide meaningful names for numeric flags used with Component Manager functions. Most applications programmers will make use of at least some of these constants.

Mac OS File System

Chapter 3, “Introduction to File Management,” documents several Mac OS API elements of interest to QuickTime 3 developers. Chapter 4, “Standard File Package,” describes how an application can use the Macintosh standard file package to manage the user interface for naming and identifying files.

You can use the following functions and data structures to retrieve user-selected files from the Mac OS file system:

- **The file system specification record** `FSSpec` provides a simple, standard format for specifying Mac OS files and directories. See “File System Specification Record” (page 153).

- **FSMakeFSSpec** (page 168) initializes a file system specification record to particular values for a file or directory.
- **Standard file reply records** identify files and folders with a file system specification record. See “Standard File Reply Records” (page 153).
- **StandardGetFile** (page 156) displays the Mac OS default Open dialog box and returns a `StandardFileReply` data structure.
- **StandardPutFile** (page 157) displays the default Save dialog box when the user is saving a file.

Memory Management

Chapters 5 and 6 describe routines in Mac OS that help you create and dispose of pointers and handles. Chapter 5, “Introduction to Memory Management,” describes memory management within an application’s memory partition; Chapter 6, “Memory Manager,” describes memory management outside the partition. The following functions are described in both chapters:

- **NewHandle** (page 287) allocates a relocatable memory block of a specified size within an application’s memory partition. To use it outside an application’s memory partition, see page 342.
- **NewHandleClear** (page 288) allocates prezeroed memory in a relocatable block of a specified size within an application’s memory partition. To use it outside an application’s memory partition, see page 344.
- **DisposeHandle** (page 289) frees a relocatable block and its master pointer for other uses within an application’s memory partition. To use it outside an application’s memory partition, see page 347.
- **NewPtr** (page 290) allocates a nonrelocatable block of memory of a specified size within an application’s memory partition. To use it outside an application’s memory partition, see page 348.
- **NewPtrClear** (page 291) allocates prezeroed memory in a nonrelocatable block of a specified size within an application’s memory partition. To use it outside an application’s memory partition, see page 350.
- **DisposePtr** (page 292) frees a nonrelocatable block for other uses within an application’s memory partition. To use it outside an application’s memory partition, see page 351.

Mac OS Events

Mac OS event management is discussed in Chapter 7, “Event Manager.” The Mac OS Event Manager returns information about retrieved events in an **event record**, a structure of type `EventRecord`. See “The Event Record” (page 414).

Window Management

Mac OS window management is discussed in Chapter 8, “Window Manager.” The function `SizeWindow` (page 528) and the data structures listed below are of interest to Windows developers working with QuickTime 3:

- The `CWindowRecord` data type defines the **color window record** for a window. See “The Color Window Record” (page 483).
- The `CWindowPeek` data type is a pointer to a **window record**. The first field in the window record is the record that describes the window’s **graphics port**. See “The Window Record” (page 488).
- The `CWindowPtr` data type is defined as a `CGrafPtr`, which is a pointer to a graphics port (in this case, the window’s graphics port). See “The Color Window Record” (page 483).
- `SizeWindow` (page 528) sets the size of a Mac OS window.

QuickDraw

Macintosh **QuickDraw** is a collection of system-level functions and data structures that other software can use to perform most operations that manipulate static two-dimensional images. Information about QuickDraw of interest to QuickTime 3 developers is contained in Chapters 9, 10, 11, and 12, as described below.

Drawing Environment

Chapter 9, “Color QuickDraw,” contains a section, “The Color Drawing Environment: Color Graphics Ports” (page 575), that describes the **color graphics port** data structure, of type `CGrafPort`. This data structure defines a complete drawing environment that determines where and how graphics operations take place.

Graphics Devices

Chapter 10, “Graphics Devices,” contains a section “Creating, Setting, and Disposing of GDevice Records” (page 695). This section tells you how to use **graphic device** records (of type `GDevice`), which store state information for video devices and offscreen graphics worlds.

QuickDraw Operations Offscreen

When using QuickDraw with QuickTime, you may want to work offscreen. Chapter 11, “Offscreen Graphics Worlds,” describes the following functions and data structures that QuickTime 3 programmers can access:

- `NewGWorld` (page 728) creates an offscreen graphics world. It uses `NewScreenBuffer` (page 733) and `NewTempScreenBuffer` (page 734) to create an offscreen `PixMap` record and allocate memory for the base address of its pixel image. It returns a pointer of type `GWorldPtr` by which your application refers to the offscreen graphics world. These types are described in “Data Structures” (page 724).
- `UpdateGWorld` (page 735) changes the pixel depth, boundary rectangle, or color table for an existing offscreen graphics world.
- `DisposeGWorld` (page 738) uses `DisposeScreenBuffer` (page 739) to dispose of all the memory allocated for an offscreen graphics world.
- `GetGWorld` (page 739) gets the current graphics port (color graphics port or offscreen graphics world) and the current `GDevice` record (see “Creating, Setting, and Disposing of GDevice Records” (page 695)).
- `SetGWorld` (page 740) changes the current graphics port (color graphics port or offscreen graphics world).
- `GetGWorldDevice` (page 741) obtains a handle to the Graphic Device record associated with an offscreen graphics world.

- `GetGWorldPixMap` (page 743) obtains the pixel map previously created for an offscreen graphics world.
- `LockPixels` (page 744) prevents the base address of an offscreen pixel image from being moved while you draw into or copy from its pixel map.
- `UnlockPixels` (page 745) frees the base address of an offscreen pixel image when you have finished drawing into or copying from an offscreen graphics world.
- `GetPixelsState` (page 748) saves current information about the memory allocated for an offscreen pixel image.
- `SetPixelsState` (page 749) restores an offscreen pixel image to the state previously saved by `GetPixelsState`.
- `GetPixBaseAddr` (page 750) obtains a pointer to an offscreen pixel map.

Several of the routines listed above expect or return values defined by the `GWorldFlags` data type, which specify a number of options for offscreen graphics worlds. It is described in “Data Structures” (page 724).

Pictures

Mac OS pictures are stored in **picture records**, data structures of type `Picture`. Chapter 12, “Pictures,” describes this data type and the function `DrawPicture` (page 793), which draws a picture stored in a picture record on any type of output device.

Component Manager

This chapter describes how you can use the Component Manager to allow your application to find and utilize various software objects (components) at run time. It also discusses how you can create your own components and how you can use the Component Manager to help manage your components. You should read this chapter if you are developing an application that uses components or if you plan to develop your own components.

The rest of this chapter

- contains a general introduction to components and the features provided by the Component Manager
- discusses how to use the facilities of the Component Manager to call components
- describes how to create a component

Several of the sections in this chapter are divided into two main topics: one describes how applications can use components, and one describes how to create your own components. If you are developing an application that uses components, you should focus on the material that describes how to use existing components—you do not need to read the material that describes how to create a component. If you are developing a component, however, you should be familiar with all the information in this chapter.

For information on a specific component, see the documentation supplied with that component. For example, for information on the components that Apple supplies with QuickTime, see *Inside Macintosh: QuickTime Components*.

Introduction to Components

A **component** is a piece of code that provides a defined set of services to one or more clients. Applications, system extensions, as well as other components can use the services of a component. A component typically provides a specific type of service to its clients. For example, a component might provide image compression or image decompression

Component Manager

capabilities; an application could call such a component, providing the image to compress, and the component could perform the desired operation and return the compressed image to the application.

Multiple components can provide the same type of service. For example, separate components might exist that can compress an image by 20 percent, 40 percent, or 50 percent, with varying degrees of fidelity. All components of the same type must support the same basic interface. This allows your application to use the same interface for any given type of component and get the same type of service, yet allows your application to obtain different levels of service.

The Component Manager provides access to components and manages them by, for example, keeping track of the currently available components and routing requests to the appropriate component.

The Component Manager classifies components by three main criteria: the type of service provided, the level of service provided, and the component manufacturer. The Component Manager uses a **component type** to identify the type of service provided by a component. Like resource types, a component type is a sequence of four characters. All components of the same component type provide the same type of services and support a common application interface. For example, all image compressor components have a component type of 'imco'. Other types of components include video digitizers, timing sources, movie controllers, and sequence capturers.

Note

Component types consisting of only lowercase characters are reserved for definition by Apple. You can define component types using other combinations of characters, but you must register any new component types with Apple's Component Registry Group (AppleLink REGISTRY). ♦

The Component Manager allows components to identify variations on the basic interface they must support by specifying a four-character **component subtype**. The value of the component subtype is meaningful only in the context of a given component type. For example, image compressor components use the component subtype to specify the compression algorithm supported by the component.

All components of a given type-subtype combination must support a common application interface. However, components that share a type-subtype specification may support routines that are not part of the basic interface defined for their type. In this manner, components can provide enhanced services to client applications while still supporting the basic application interface.

Finally, the Component Manager allows components to have a four-character manufacturer code that identifies the manufacturer of the component. You must register your component with Apple's Component Registry Group to receive a manufacturer code for your component. The manufacturer code allows applications to further distinguish between components of the same type-subtype.

About the Component Manager

The Component Manager provides services that allow applications to obtain run-time location of and access to functional objects (in much the same way that the Resource Manager allows applications that are running to access data objects dynamically).

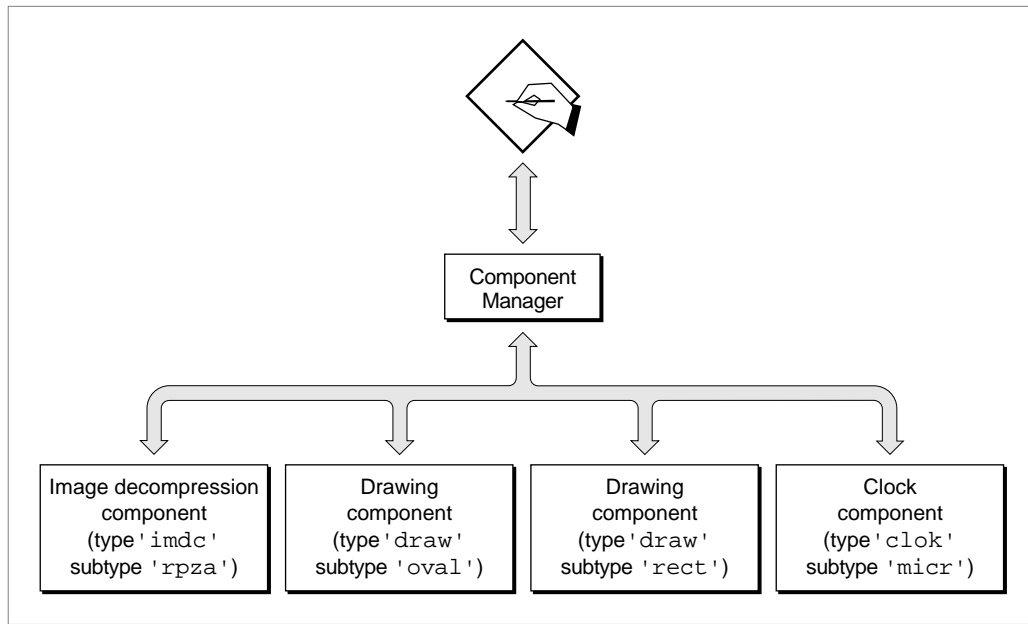
The Component Manager creates an interface between components and clients, which can be applications, other components, system extensions, and so on. Instead of implementing support for a particular data format, protocol, or model of a device, you can use a standard interface through which your application communicates with all components of a given type. You can then use the Component Manager to locate and communicate with components of that type. Those components, in turn, provide the appropriate services to your client application.

Given a particular component type, the Component Manager can locate and query all components of that type. You can find out how many components of a specific type are available and you can get further details about a component's capabilities without having to open it first. For each component, the Component Manager keeps track of many characteristics, including its name, icon, and information string.

For example, components of type `'imdc'` provide image decompression services. All components of type `'imdc'` share a common application interface, but each image decompressor component may support a unique compression technique or take advantage of a special hardware implementation. Individual components may support additions to the defined application interface, as long as they support the common routines. Any algorithm-dependent or implementation-dependent variations of the general decompression interface can be implemented by each `'imdc'` component as extensions to the basic interface.

Figure 2-1 shows the relationship between an application, the Component Manager, and several components. Applications and other clients use the Component Manager to access components. In this figure, four components are available to the application: an image decompression component (of type `'imdc'`), two drawing components (of type `'draw'`), and a clock component (of type `'clock'`). Note that the two drawing components have different subtypes: `'oval'` and `'rect'`. The drawing component with subtype `'oval'` draws ovals, and the drawing component with subtype `'rect'` draws rectangles.

Figure 2-1 The relationship between an application, the Component Manager, and components



The Component Manager allows a single component to serve multiple client applications at the same time. Each client application has a unique access path to the component. These access paths are called **component connections**. You identify a component connection by specifying a **component instance**. The Component Manager provides this component instance to your application when you open a connection to a component. The component maintains separate status information for each open connection.

For example, multiple applications might each open a connection to an image decomposition component. The Component Manager routes each application request to the component instance for that connection. Because a component can maintain separate storage for each connection, application requests do not interfere with each other and each application has full access to the services provided by the component. (See Figure 2-2 on page 2-62 for an illustration of multiple applications using the services of the same component.)

Using the Component Manager

This section describes how you can use the Component Manager to

- gain access to components
- locate components and take advantage of their services

Component Manager

- get information about a component
- close a connection to a component

The Component Manager is available in System 7.1 or later and may be present in System 7. To determine whether the Component Manager is available, call the `Gestalt` function with the `gestaltComponentMgr` selector and check the value of the `response` parameter.

```
CONST
    gestaltComponentMgr    = 'cpnt';
```

The `Gestalt` function returns in the `response` parameter a 32-bit value indicating the version of the Component Manager that is installed. Version 3 and above supports automatic version control, the unregister request, and icon families. You should test the version number before using any of these features.

This section presents several examples demonstrating how to use components and the Component Manager. All of these examples use the services of a drawing component—a simple component that draws an object of a particular shape on the screen. Drawing components have a component type of `'draw'`. The component subtype value indicates the type of object the component draws. For example, a drawing component that draws an oval has a component subtype of `'oval'`. For information on creating your own components and for listings that show the code for a drawing component, see “Creating Components” beginning on page 2-41.

Opening Connections to Components

When your application requires the services of a component, you typically perform these steps:

- open a connection to the desired component
- use the services of the component
- close the connection to the component

The following sections describe each of these steps in more detail.

Opening a Connection to a Default Component

Your application must use the Component Manager to gain access to a component. The first step is to locate an appropriate component. You can locate the component yourself, or you can allow the Component Manager to locate a suitable component for you. Your application then opens a connection to that component. Once you have opened a connection to a component, you can use the services provided by that component. When you have finished using the component, you should close the connection.

If you are interested only in using a component of a particular type-subtype and you do not need to specify any other characteristics of the component, use the `OpenDefaultComponent` function and specify only the component type and subtype—the Component Manager then selects a component for you and opens a connection to

Component Manager

that component. This is the easiest technique for opening a component connection. The `OpenDefaultComponent` function searches its list of available components and attempts to open a connection to a component with the specified type and subtype. If more than one component of the specified type and subtype is available, `OpenDefaultComponent` selects the first one in the list. If successful, the `OpenDefaultComponent` function returns a component instance that identifies your connection to the component. You can then use that connection to employ the services of the selected component.

This code demonstrates the use of the `OpenDefaultComponent` function. The code opens a connection to a component of type 'draw' and subtype 'oval'—a drawing component that draws an oval.

VAR

```
aDrawOvalComp: ComponentInstance;

aDrawOvalComp := OpenDefaultComponent('draw', 'oval');
```

If it cannot find or open a component of the specified type-subtype, the `OpenDefaultComponent` function returns a function result of `NIL`.

To open a connection to a component with a specific type-subtype-manufacturer code or with other specified characteristics, first use the `FindNextComponent` function to find the desired component, then open the component using the `OpenComponent` function. These operations are described in the next two sections.

Finding a Specific Component

If you are interested in asserting greater control over the selection of a component, you can use the Component Manager to find a component that provides a specified service. For example, you can use the `FindNextComponent` function in a loop to retrieve information about all the components that are registered on a given computer. Each time you call this function, the Component Manager returns information about a single component. You can obtain a count of all the components on a given computer by calling the `CountComponents` function. Both of these functions allow you to specify search criteria, for example, by component type and subtype, or by manufacturer. By using these criteria to narrow your search, you can quickly and easily find a component that meets your needs.

You specify the search criteria for the component using a component description record. A component description record is defined by the `ComponentDescription` data type. For more information on the fields of this record, see “The Component Description Record” beginning on page 2-65.

TYPE

```
ComponentDescription =
  RECORD
    componentType:      OType;      {type}
    componentSubType:   OType;      {subtype}
```

Component Manager

```

        componentManufacturer:  OSType;      {manufacturer}
        componentFlags:         LongInt;     {control flags}
        componentFlagsMask:     LongInt;     {mask for flags}
    END;

```

By default, the Component Manager considers all fields of the component description record when performing a search. Your application can override the default behavior of which fields the Component Manager considers for a search. Specify 0 in any field of the component description record to prevent the Component Manager from considering the information in that field when performing the search.

Listing 2-1 shows an application-defined procedure, `MyFindVideoComponent`, that fills out a component description record to specify the search criteria for the desired component. The `MyFindVideoComponent` procedure then uses the `FindNextComponent` function to return the first component with the specified characteristics—in this example, any component with the type `VideoDigitizerComponentType`.

Listing 2-1 Finding a component

```

PROCEDURE MyFindVideoComponent(VAR videoCompID: Component);
VAR
    videoDesc: ComponentDescription;
BEGIN
    {find a video digitizer component}
    videoDesc.componentType := VideoDigitizerComponentType;
    videoDesc.componentSubType := OSType(0);      {any subtype}
    videoDesc.componentManufacturer:= OSType(0); {any manufacturer}
    videoDesc.componentFlags := 0;
    videoDesc.componentFlagsMask := 0;
    videoCompID := FindNextComponent(Component(0), videoDesc);
END;

```

The `FindNextComponent` function requires two parameters: a value that indicates which component to begin the search with and a component description record. You can specify 0 in the first parameter to start the search at the beginning of the component list. Alternatively, you can specify a component identifier obtained from a previous call to `FindNextComponent`.

The `FindNextComponent` function returns a component identifier to your application. The returned component identifier identifies a given component to the Component Manager. You can use this identifier to retrieve more information about the component or to open a connection to the component. The next two sections describe these tasks.

Opening a Connection to a Specific Component

You can open a connection to a specific component by calling the `OpenComponent` function (alternatively, you can use the `OpenDefaultComponent` function, as discussed in “Opening a Connection to a Default Component” on page 2-35). Your application must provide a component identifier to the `OpenComponent` function. You get a component identifier from the `FindNextComponent` function, as described in the previous section.

The `OpenComponent` function returns a component instance that identifies your connection to the component. Listing 2-2 shows how to use the `OpenComponent` function to gain access to a specific component. The application-defined procedure `MyGetComponent` uses the `MyFindVideoComponent` procedure (defined in Listing 2-1) to find a video digitizer component and then opens the component.

Listing 2-2 Opening a specific component

```
PROCEDURE MyGetComponent
                                (VAR videoCompInstance: ComponentInstance);
VAR
    videoCompID:      Component;
BEGIN
    {first find a video digitizer component}
    MyFindVideoComponent(videoCompID);
    {now open it}
    IF videoCompID <> NIL THEN
        videoCompInstance := OpenComponent(videoCompID);
    END;
```

Getting Information About a Component

You can use the `GetComponentInfo` function to retrieve information about a component, including the component name, icon, and other information. Listing 2-3 shows an application-defined procedure that retrieves information about a video digitizer component.

Listing 2-3 Getting information about a component

```
PROCEDURE MyGetCompInfo (compName, compInfo, compIcon: Handle;
                        VAR videoDesc: ComponentDescription);
VAR
    videoCompID:      Component;
    myErr:            OSErr;
BEGIN
    {first find a video digitizer component}
```

Component Manager

```

MyFindVideoComponent(videoCompID);
{now get information about it}
IF videoCompID <> NIL THEN
    myErr := GetComponentInfo(videoCompID, videoDesc, compName,
                              compInfo, compIcon);
END;

```

You specify the component in the first parameter to `GetComponentInfo`. You specify the component using either a component identifier (obtained from `FindNextComponent` or `RegisterComponent`) or a component instance (obtained from `OpenDefaultComponent` or `OpenComponent`).

The `GetComponentInfo` function returns information about the component in the second through fifth parameters of the function. The `GetComponentInfo` function returns information about the component (such as its type, subtype, and manufacturer) in a component description record. The function also returns the component name, icon, and other information through handles. You must allocate these handles before calling `GetComponentInfo`. (Alternatively, you can specify `NIL` in the `compName`, `compInfo`, and `compIcon` parameters if you do not want the information returned.) The icon returned in the `compIcon` parameter is a handle to a black-and-white icon. If a component has an icon family, you can retrieve a handle to its icon suite using `GetComponentIconSuite`.

Using a Component

Once you have established a connection to a component, you can use its services.

Each time you call a component routine, you must specify the component instance that identifies your connection and provide any other parameters as required by the routine.

For example, Listing 2-4 illustrates the use of a drawing component. The application-defined procedure establishes a connection to a drawing component, calls the component's `DrawerSetup` function to establish the rectangle in which to draw the desired object, and then draws the object using the `DrawerDraw` function.

Listing 2-4 Using a drawing component

```

PROCEDURE MyDrawAnOval (VAR aDrawOvalComp: ComponentInstance);
VAR
    r:           Rect;
    result:      ComponentResult;
BEGIN
    {open a connection to a drawing component}
    aDrawOvalComp := OpenDefaultComponent('draw', 'oval');
    IF aDrawOvalComp <> NIL THEN
        BEGIN

```

Component Manager

```

        SetRect(r, 40, 40, 80, 80);
        {set up rectangle for oval}
        result := DrawerSetup(aDrawOvalComp, r);
        IF result = noErr THEN
            result := DrawerDraw(aDrawOvalComp); {draw oval}
        END;
    END;
END;

```

If you specify an invalid connection as a parameter to a component routine, the Component Manager sets the function result of the component routine to `badComponentInstance`.

Each component type supports a defined set of functions. You must refer to the appropriate documentation for a description of the functions supported by a component. You also need to refer to the component's documentation for information on the appropriate interface files that you must include to use the component (the interface files for the drawing component are shown beginning on page 2-56). The components that Apple provides with QuickTime are described in *Inside Macintosh: QuickTime Components*. As an example, drawing components support the following functions:

```

FUNCTION DrawerSetup(myInstance: ComponentInstance;
                    VAR r: Rect): ComponentResult;
FUNCTION DrawerClick(myInstance: ComponentInstance;
                    p: Point): ComponentResult;
FUNCTION DrawerMove (myInstance: ComponentInstance; x: Integer;
                    y: Integer): ComponentResult;
FUNCTION DrawerDraw (myInstance: ComponentInstance)
                    : ComponentResult;
FUNCTION DrawerErase(myInstance: ComponentInstance)
                    : ComponentResult;

```

Closing a Connection to a Component

When you finish using a component, you must close your connection to that component. Use the `CloseComponent` function to close the connection. For example, this code calls the application-defined procedure `MyDrawAnOval` (see Listing 2-4), which opens a connection to a drawing component and uses that component to draw an oval. This code closes the oval drawer component after it is finished using it.

```

VAR
    aDrawOvalComp: ComponentInstance;
    result:         OSErr;

MyDrawAnOval(aDrawOvalComp);    {open component and draw an oval}
result := DrawerErase(aDrawOvalComp); {erase the oval}
result := CloseComponent(aDrawOvalComp); {close the component}

```


Creating Components

This section describes how to create a component and how your component interacts with the Component Manager. This section also describes many of the routines that the Component Manager provides to help you manage your component. If you are developing a component, you should read the material in this section.

If you are developing an application that uses components, you may find this material interesting, but you do not need to be familiar with it. You should read the preceding section, “Using the Component Manager,” and then use the “Component Manager Reference” section as needed.

This section discusses how you can

- structure your component
- respond to requests from the Component Manager
- define the functions that applications may call to request service from your component
- manage your component with the help of the Component Manager
- make your component available for use by applications

This section presents several examples demonstrating how to create components and register them with the Component Manager. All of these examples are based on a “drawing component”—a simple component that draws an object of a particular shape on the screen. This section includes the code for a drawing component.

The Structure of a Component

Every component must have a single entry point that returns a value of type `ComponentResult` (a long integer). Whenever the Component Manager receives a request for your component, it calls your component’s entry point and passes any parameters, along with information about the current connection, in a component parameters record. The Component Manager also passes a handle to the global storage (if any) associated with that instance of your component.

When your component receives a request, it should examine the parameters to determine the nature of the request, perform the appropriate processing, set an error code if necessary, and return an appropriate function result to the Component Manager.

Component Manager

The component parameters record is defined by a data structure of type `ComponentParameters`.

```

TYPE ComponentParameters =
    PACKED RECORD
        flags:      Char;           {reserved}
        paramSize:  Char;           {size of parameters}
        what:       Integer;        {request code}
        params:     ARRAY[0..0] OF LongInt; {actual parameters}
    END;

```

The `what` field contains a value that specifies the type of request. Negative values are reserved for definition by Apple. You can use values greater than or equal to 0 to define other requests that are supported by your component. Follow these guidelines when defining your request codes: request codes between 0 and 256 are reserved for definition by components of a given type and a given type-subtype. Use request codes greater than 256 for requests that are unique to your component. For example, a certain component of a certain type-subtype might support values 0 through 5 as requests that are supported by all components of that type, values 128 through 140 as requests that are supported by all components of that given type-subtype, and values 257 through 260 as requests supported only by that component.

Table 2-1 shows the request codes defined by Apple and the actions your component should take upon receiving them. Note that four of the request codes—open, close, can do, and version—are required. Your component must respond to these four request codes. These request codes are described in greater detail in “Handling Requests for Service” beginning on page 2-46.

Table 2-1 Request codes

Request code	Action your component should perform	Required
<code>kComponentOpenSelect</code>	Open a connection	Yes
<code>kComponentCloseSelect</code>	Close an open connection	Yes
<code>kComponentCanDoSelect</code>	Determine whether your component supports a particular request	Yes
<code>kComponentVersionSelect</code>	Return your component's version number	Yes
<code>kComponentRegisterSelect</code>	Determine whether your component can operate in the current environment	No

Table 2-1 Request codes (continued)

Request code	Action your component should perform	Required
kComponentTargetSelect	Call another component whenever it would call itself (as a result of your component being used by another component)	No
kComponentUnregisterSelect	Perform any operations that are necessary as a result of your component being unregistered	No

The example drawing component (shown in Listing 2-5 on page 2-44) supports the four required request codes, and in addition supports the request codes that are required for all components of the type 'draw'. All drawing components must support these request codes:

```
CONST
    kDrawerSetUpSelect      = 0;  {set up drawing region}
    kDrawerDrawSelect      = 1;  {draw the object}
    kDrawerEraseSelect     = 2;  {erase the object}
    kDrawerClickSelect     = 3;  {determine if cursor is }
                                { inside of the object}
    kDrawerMoveSelect      = 4;  {move the object}
```

The `params` field of the component parameters record is an array that contains the parameters specified by the application that called your component. You can directly extract the parameters from this array, or you can use the `CallComponentFunction` or `CallComponentFunctionWithStorage` function to extract the parameters from this array and pass these parameters to a subroutine of your component (see page 2-91 and page 2-92 for more information about these functions).

Listing 2-5 shows the structure of a drawing component—a simple component that draws an object on the screen. The component subtype of a drawing component indicates the type of object the component draws. This particular drawing component is of the subtype 'oval'; it draws oval objects.

Whenever an application calls your component, the Component Manager calls your component's main entry point (for example, the `OvalDrawer` function). This entry point must be the first function in the component's code segment.

As previously described, the Component Manager passes two parameters to your component: a component parameters record and a parameter of type `Handle`. The parameters specified by the calling application are contained in the component parameters record. Your component can access the parameters directly from this record. Alternatively, as shown in Listing 2-5, you can use Component Manager routines to extract the parameters from this array and invoke a subroutine of your component. By taking advantage of these routines, you can simplify the structure of your component code.

Component Manager

The `OvalDrawer` function first examines the value of the `what` field of the component parameters record. The `what` field contains the request code. The `OvalDrawer` function performs the action specified by the request code. The `OvalDrawer` function uses a number of subroutines to carry out the desired action. It uses the Component Manager routines `CallComponentFunction` and `CallComponentFunctionWithStorage` to extract the parameters from the component parameters record and to call the specified component's subroutine with these parameters.

For example, when the drawing component receives the request code `kComponentOpenSelect`, it calls the function `CallComponentFunction`. It passes the component parameters record and a pointer to the component's `OvalOpen` subroutine as parameters to `CallComponentFunction`. This function extracts the parameters and calls the `OvalOpen` function. The `OvalOpen` function allocates memory for this instance of the component. Your component can allocate memory to hold global data when it receives an open request. To do this, allocate the memory and then call the `SetComponentInstanceStorage` function. This function associates the allocated memory with the current instance of your component. The next time this instance of your component is called, the Component Manager passes a handle to your previously allocated memory in the `storage` parameter. For additional information on handling the open request, see “Responding to the Open Request” on page 2-47.

When the drawing component receives the drawing setup request (indicated by the `kDrawerSetupSelect` constant), it calls the Component Manager function `CallComponentFunctionWithStorage`. Like `CallComponentFunction`, this function extracts the parameters and calls the specified subroutine (`OvalSetup`). The `CallComponentFunctionWithStorage` function also passes as a parameter to the subroutine a handle to the memory associated with this instance of the component. The `OvalSetup` subroutine can use this memory as needed. For additional information on handling the drawing setup request, see “Responding to Component-Specific Requests” on page 2-54.

Listing 2-5 A drawing component for ovals

```

UNIT Ovals;
INTERFACE
{include a USES statement if required}

FUNCTION OvalDrawer (params: ComponentParameters;
                    storage: Handle): ComponentResult;

IMPLEMENTATION

CONST
    kOvalDrawerVersion      = 0;  {version number of this component}

    kDrawerSetUpSelect      = 0;  {set up drawing region}

```

Component Manager

```

kDrawerDrawSelect      = 1;  {draw the object}
kDrawerEraseSelect     = 2;  {erase the object}
kDrawerClickSelect     = 3;  {determine if cursor is }
                           { inside of the object}
kDrawerMoveSelect      = 4;  {move the object}

TYPE
  GlobalsRecord =
    RECORD
      bounds:          Rect;
      boundsRgn:       RgnHandle;
      self:            ComponentInstance;
    END;
  GlobalsPtr      = ^GlobalsRecord;
  GlobalsHandle   = ^GlobalsPtr;

{any subroutines used by the component go here}

FUNCTION OvalDrawer (params: ComponentParameters;
                    storage: Handle): ComponentResult;
BEGIN
  {perform action corresponding to request code}
  IF params.what < 0 THEN    {handle the required request codes}
    CASE (params.what) OF
      kComponentOpenSelect:
        OvalDrawer := CallComponentFunction(params,
                                              ComponentRoutine(@OvalOpen));
      kComponentCloseSelect:
        OvalDrawer := CallComponentFunctionWithStorage(storage, params,
                                              ComponentRoutine(@OvalClose));
      kComponentCanDoSelect:
        OvalDrawer := CallComponentFunction(params,
                                              ComponentRoutine(@OvalCanDo));
      kComponentVersionSelect:
        OvalDrawer := kOvalDrawerVersion;
    OTHERWISE
      OvalDrawer := badComponentSelector;
    END {of CASE}
  ELSE                      {handle component-specific request codes}
    CASE (params.what) OF
      kDrawerSetupSelect:
        OvalDrawer := CallComponentFunctionWithStorage
                      (storage, params,
                      ComponentRoutine(@OvalSetup));

```

Component Manager

```

kDrawerDrawSelect:
    OvalDrawer := CallComponentFunctionWithStorage
                    (storage, params,
                    ComponentRoutine(@OvalDraw));

kDrawerEraseSelect:
    OvalDrawer := CallComponentFunctionWithStorage
                    (storage, params,
                    ComponentRoutine(@OvalErase));

kDrawerClickSelect:
    OvalDrawer := CallComponentFunctionWithStorage
                    (storage, params,
                    ComponentRoutine(@OvalClick));

kDrawerMoveSelect:
    OvalDrawer := CallComponentFunctionWithStorage
                    (storage, params,
                    ComponentRoutine(@OvalMoveTo));

OTHERWISE
    OvalDrawer := badComponentSelector;
END; {of CASE}
END; {of OvalDrawer}

END.

```

The next section describes how your component should respond to the required request codes. Following sections provide more information on

- defining your component's interfaces
- registering your component
- how to store your component in a component resource file

Handling Requests for Service

Whenever an application requests services from your component, the Component Manager calls your component and passes two parameters: the application's parameters in a component parameters record and a handle to the memory associated with the current connection. The component parameters record also contains information identifying the nature of the request.

There are two classes of requests: requests that are defined by the Component Manager and requests that are defined by your component. The Component Manager defines seven request codes: open, close, can do, version, register, unregister, and target. All components must support open, close, can do, and version requests. The register, unregister, and target requests are optional. Apple reserves all negative request codes for definition by the Component Manager. You are free to assign request codes greater than or equal to 0 to the functions supported by a component whose interface you have

Component Manager

defined. (However, request codes between 0 and 256 are reserved for definition by components of a given type and a given type-subtype. Request codes greater than 256 are available for requests that are unique to your component.)

You can refer to the standard request codes with these constants.

```
CONST kComponentOpenSelect      = -1; {open request}
      kComponentCloseSelect     = -2; {close request}
      kComponentCanDoSelect     = -3; {can do request}
      kComponentVersionSelect   = -4; {version request}
      kComponentRegisterSelect  = -5; {register request}
      kComponentTargetSelect    = -6; {target request}
      kComponentUnregisterSelect = -7; {unregister request}
```

The following sections discuss what your component must do when it receives these Component Manager requests.

Responding to the Open Request

The Component Manager issues an open request to your component whenever an application or any other client tries to open a connection to your component by calling the `OpenComponent` (or `OpenDefaultComponent`) function. The open request allows your component to establish the environment to support the new connection. Your component must support this request.

Your component should perform the necessary processing to establish the new connection. At a minimum, you must allocate the memory for any global data for the connection. Be sure to allocate this memory in the current heap zone, not in the system heap. You should call the `SetComponentInstanceStorage` procedure to inform the Component Manager that you have allocated memory. The Component Manager stores a handle to the memory and provides that handle to your component as a parameter in subsequent requests.

You may also want to open and read data from your component's resource file—if you do so, use the `OpenComponentResFile` function to open the file and be sure to close the resource file before returning.

If your component uses the services of other components, open connections to them when you receive the open request.

Once you have successfully set up the connection, set your component's function result to 0 and return to the Component Manager.

You can also refuse the connection. If you cannot successfully establish the environment for a connection (for example, there is insufficient memory to support the connection, or required hardware is unavailable), you can refuse the connection by setting the component's function result to a nonzero value. You can also use the open request as an opportunity to restrict the number of connections your component can support.

If your application is registered globally, you should also set the A5 world for your component in response to the open request. You can do this using the

Component Manager

SetComponentInstanceA5 procedure. (See page 2-96 for information on this procedure.)

The Component Manager sets these fields in the component parameters record that it provides to your component on an open request:

Field descriptions

what	The Component Manager sets this field to kComponentOpenSelect.
params	The first entry in this array contains the component instance that identifies the new connection.

Listing 2-6 shows the subroutine that handles the open request for the drawing component. Note that your component can directly access the parameters from the component parameters record, or use subroutines and the CallComponentFunction and CallComponentFunctionWithStorage functions to extract the parameters for you (see Listing 2-5 on page 2-44). The code in this chapter takes the second approach.

The OvalOpen function allocates memory to hold global data for this instance of the component. It calls the SetComponentInstanceStorage function so that the Component Manager can associate the allocated memory with this instance of the component. The Component Manager passes a handle to this memory in subsequent calls to this instance of the component.

Listing 2-6 Responding to an open request

```

FUNCTION OvalOpen (self: ComponentInstance): ComponentResult;
VAR
    myGlobals: GlobalsHandle;
BEGIN
    {allocate storage}
    myGlobals :=
        GlobalsHandle(NewHandleClear(sizeof(GlobalsRecord)));
    IF myGlobals = NIL THEN
        OvalOpen := MemError
    ELSE
        BEGIN
            myGlobals^.self := self;
            myGlobals^.boundsRgn := NewRgn;
            SetComponentInstanceStorage(myGlobals^.self,
                                       Handle(myGlobals));
            {if your component is registered globally, set }
            { its A5 world before returning}
            OvalOpen := noErr;
        END;
    END;
END;

```


Responding to the Close Request

The Component Manager issues a close request to your component when a client application closes its connection to your component by calling the `CloseComponent` function. Your component should dispose of the memory associated with the connection. Your component must support this request. Your component should also close any files or connections to other components that it no longer needs.

The Component Manager sets these fields in the component parameters record that it provides to your component on a close request:

Field descriptions

<code>what</code>	The Component Manager sets this field to <code>kComponentCloseSelect</code> .
<code>params</code>	The first entry in this array contains the component instance that identifies the open connection.

Listing 2-7 shows the subroutine that handles the close request for the drawing component (as defined in Listing 2-5 on page 2-44). The `OvalClose` function closes the open connection. The drawing component uses the `CallComponentFunctionWithStorage` function to call the `OvalClose` function (see Listing 2-5). Because of this, in addition to the parameters specified in the component parameters record, the Component Manager also passes to the `OvalClose` function a handle to the memory associated with the component instance.

Listing 2-7 Responding to a close request

```
FUNCTION OvalClose (globals: GlobalsHandle;
                  self: ComponentInstance): ComponentResult;
BEGIN
  IF globals <> NIL THEN
    BEGIN
      DisposeRgn(globals^^.boundsRgn);
      DisposeHandle(Handle(globals));
    END;
    OvalClose := noErr;
  END;
```

IMPORTANT

When responding to a close request, you should always test the handle passed to your component against `NIL` because it is possible for your close request to be called with a `NIL` handle in the `storage` parameter. For example, you can receive a `NIL` handle if your component returns a nonzero function result in response to an open request. ▲

Responding to the Can Do Request

The Component Manager issues a can do request to your component when an application calls the `ComponentFunctionImplemented` function to determine whether your component supports a given request code. Your component must support this request.

Set your component's function result to 1 if you support the request code; otherwise, set your function result to 0.

The Component Manager sets these fields in the component parameters record that it provides to your component on a can do request:

Field descriptions

<code>what</code>	The Component Manager sets this field to <code>kComponentCanDoSelect</code> .
<code>params</code>	The first entry in this array contains the request code as an integer value.

Listing 2-8 shows the subroutine that handles the can do request for the drawing component (as defined in Listing 2-5 on page 2-44). The `OvalCanDo` function examines the specified request code and compares it with the request codes that it supports. It returns a function result of 1 if it supports the request code; otherwise, it returns 0.

Listing 2-8 Responding to the can do request

```
FUNCTION OvalCanDo (selector: Integer): ComponentResult;
BEGIN
    IF ((selector >= kComponentVersionSelect) AND
        (selector <= kDrawerMoveSelect)) THEN
        OvalCanDo := 1                {valid request}
    ELSE
        OvalCanDo := 0;              {invalid request}
END;
```

Responding to the Version Request

The Component Manager issues a version request to your component when an application calls the `GetComponentVersion` function to retrieve your component's version number. Your component must support this request.

In response to a version request, your component should return its version number as its function result. Use the high-order 16 bits to represent the major version and the low-order 16 bits to represent the minor version. The major version should represent the component specification level; the minor version should represent your implementation's version number.

Component Manager

If the Component Manager supports automatic version control (a feature available in version 3 and above of the manager), it automatically resolves conflicts between different versions of the same component. For more information on this feature, see the next section, “Responding to the Register Request.”

The Component Manager sets only the `what` field in the component parameters record that it provides to your component on a version request:

Field description

<code>what</code>	The Component Manager sets this field to <code>kComponentVersionSelect</code> .
-------------------	---

Listing 2-5 on page 2-44 shows how the drawing component handles the version request. It simply returns its version number as its function result.

Responding to the Register Request

The Component Manager may issue a register request when your component is registered. This request gives your component an opportunity to determine whether it can operate in the current environment. For example, your component might use the register request to verify that a specific piece of hardware is available on the computer. This is an optional request—your component is not required to support it.

The Component Manager issues this request only if you have set the `cmpWantsRegisterMessage` flag to 1 in the `componentFlags` field of your component’s component description record (see “Data Structures for Components” beginning on page 2-80 for more information about the component description record).

Your component should not normally allocate memory in response to the register request. The register request is provided so that your application can determine whether it should be registered and made available to any clients. Once a client attempts to connect to your component, your component receives an open request, at which time it can allocate any required memory. Because your component might not be opened during a particular session, following this guideline allows other applications to make use of memory that isn’t currently needed by your component.

If you want the Component Manager to provide automatic version control (a feature available in version 3 and above of the manager), your component can specify the `componentDoAutoVersion` flag in the optional extension to the component resource. If you specify this flag, the Component Manager registers your component only if there is no later version available. If an older version is already registered, the Component Manager unregisters it. If a newer version of the same component is registered after yours, the Component Manager automatically unregisters your component. You can use this automatic version control feature to make sure that the most recent version of your component is registered, regardless of the number of versions that are installed.

Set your function result to `TRUE` to indicate that you do not want your component to be registered; otherwise, set the function result to `FALSE`.

Component Manager

The Component Manager sets only the `what` field in the component parameters record that it provides to your component on a register request:

Field description

<code>what</code>	The Component Manager sets this field to <code>kComponentRegisterSelect</code> .
-------------------	--

If you request that your component receive a register request, the Component Manager actually sends your component a series of three requests: an open request, then the register request, followed by a close request.

For more information about the process the Component Manager uses to register components, see “Registering a Component” on page 2-58.

Responding to the Unregister Request

The unregister request is supported only in version 3 and above of the Component Manager. If your component specifies the `componentWantsUnregister` flag in the `componentRegisterFlags` field of the optional extension to the component resource, the Component Manager may issue an unregister request when your component is unregistered. This request gives your component an opportunity to perform any clean-up operations, such as resetting the hardware. This is an optional request—your component is not required to support it.

Return any error information as your component’s function result.

The Component Manager sets only the `what` field in the component parameters record that it provides to your component on an unregister request:

Field description

<code>what</code>	The Component Manager sets this field to <code>kComponentUnregisterSelect</code> .
-------------------	--

If you have specified that your component should not receive a register request, then your component does not receive an unregister request if it has not been opened. However, if a client opens and closes your component, and then later the Component Manager unregisters your component, the Component Manager does send your component an unregister request (in a series of three requests: open, unregister, close).

If you have specified that your component should receive a register request, when your component is registered the Component Manager sends your component a series of three requests: an open request, then the register request, followed by a close request. In this situation, even if your component is not opened by a client, the Component Manager sends your component an unregister request when it unregisters your component.

For more information about the `componentWantsUnregister` flag, see “Resources” beginning on page 2-107.

Responding to the Target Request

The Component Manager issues a target request to inform an instance of your component that it has been targeted by another component. The component that targets

Component Manager

another component instance may also choose to first capture the component, but it is not necessary to do so. Thus, a component can choose to

- capture a component and target an instance of it
- capture a component without targeting any instance of it
- target a component instance without capturing the component

To first capture another component, the capturing component calls the `CaptureComponent` function. When a component is captured, the Component Manager removes it from the list of available components. This makes the captured component available only to the capturing component and to any clients currently connected to it. Typically, a component captures another component when it wants to override one or more functions of the other component.

After calling the `CaptureComponent` function, the capturing component can choose to target a particular instance of the component. However, a component can capture another component without targeting it.

A component uses the `ComponentSetTarget` function to send a target request to a specific component instance. After receiving a target request, whenever the targeted component instance would call itself (that is, call any of its defined functions), instead it should always call the component that targeted it.

For example, a component called `NewMath` might first capture a component called `OldMath`. `NewMath` does this by using `FindNextComponent` to get a component identifier for `OldMath`. `NewMath` then calls `CaptureComponent` to remove `OldMath` from the list of available components. At this point, no other clients can access `OldMath`, except for those clients previously connected to it.

`NewMath` might then call `ComponentSetTarget` to target a particular component instance of `OldMath`. The `ComponentSetTarget` function sends a target request to the specified component instance. When `OldMath` receives a target request, it saves the component instance of the component that targeted it. When `OldMath` receives a request, it processes it as usual. However, whenever `OldMath` calls one of its defined functions: in its defined API, it calls `NewMath` instead. (Suppose `OldMath` provides request codes for these functions: `DoMultiply`, `DoAdd`, `DoDivide`, and `DoSubtract`. If `OldMath`'s `DoMultiply` function calls its own `DoAdd` function, then `OldMath` calls `NewMath` to perform the addition.)

The target request is an optional request—your component is not required to support it.

The Component Manager sets these fields in the component parameters record that it provides to your component on a target request:

Field descriptions

<code>what</code>	The Component Manager sets this field to <code>kComponentTargetSelect</code> .
<code>params</code>	The first entry in this array contains the component instance that identifies the component issuing the target request.

Responding to Component-Specific Requests

When your component receives a component-specific request, it should handle the request as appropriate. For example, the drawing component responds to five component-specific requests: setup, draw, erase, click, and move to. See Listing 2-5 on page 2-44 for the code that defines the drawing component's entry point. The drawing component uses `CallComponentFunctionWithStorage` to extract the parameters and call the appropriate subroutine.

Listing 2-9 shows the drawing component's `OvalSetup` function. This function sets up the data structures that must be in place before drawing the oval.

Listing 2-9 Responding to the setup request

```
FUNCTION OvalSetup (globals: GlobalsHandle;
                   boundsRect: Rect): ComponentResult;
VAR
    ignoreErr: ComponentResult;
BEGIN
    globals^^.bounds := boundsRect;
    OpenRgn;
    ignoreErr := OvalDraw(globals);
    CloseRgn(globals^^.boundsRgn);
    OvalSetup := noErr;
END;
```

Listing 2-10 shows the drawing component's `OvalDraw` function. This function draws an oval in the previously allocated region.

Listing 2-10 Responding to the draw request

```
FUNCTION OvalDraw (globals: GlobalsHandle): ComponentResult;
BEGIN
    FrameOval(globals^^.bounds);
    OvalDraw := noErr;
END;
```

Listing 2-11 shows the drawing component's `OvalErase` function. This function erases an oval.

Listing 2-11 Responding to the erase request

```

FUNCTION OvalErase (globals: GlobalsHandle): ComponentResult;
BEGIN
    EraseOval(globals^^.bounds);
    OvalErase := noErr;
END;

```

Listing 2-12 shows the drawing component's `OvalClick` function. This function determines whether the given point is within the oval. If so, the function returns 1; otherwise, it returns 0. Because the `OvalClick` function returns information other than error information as its function result, `OvalClick` sets any error information using `SetComponentInstanceError`.

Listing 2-12 Responding to the click request

```

FUNCTION OvalClick (globals: GlobalsHandle; p: Point)
                    : ComponentResult;
BEGIN
    IF PtInRgn(p, globals^^.boundsRgn) THEN
        OvalClick := 1
    ELSE
        OvalClick := 0;
        SetComponentInstanceError(globals^^.self, noErr);
    END;
END;

```

Listing 2-13 shows the drawing component's `OvalMoveTo` function. This function moves the oval's coordinates to the specified location. Note that this function does not erase or draw the oval; the calling application is responsible for issuing the appropriate requests. For example, the calling application can issue requests to draw, erase, move to, and draw—to draw the oval in one location, then erase the oval, move it to a new location, and finally draw the oval in its new location.

Listing 2-13 Responding to the move to request

```

FUNCTION OvalMoveTo (globals: GlobalsHandle; x, y: Integer)
    : ComponentResult;

VAR
    r: Rect;
BEGIN
    r := globals^^.bounds;
    x := x - (r.right + r.left) DIV 2;
    y := y - (r.bottom + r.top) DIV 2;
    OffsetRect(globals^^.bounds, x, y);
    OffsetRgn(globals^^.boundsRgn, x, y);
    OvalMoveTo := noErr;
END;

```

Reporting an Error Code

The Component Manager maintains error state information for all currently active connections. In general, your component returns error information in its function result; a nonzero function result indicates an error occurred, and a function result of 0 indicates the request was successful. However, some requests require that your component return other information as its function result. In these cases, your component can use the `SetComponentInstanceError` procedure to report its latest error state to the Component Manager. You can also use this procedure at any time during your component's execution to report an error.

Defining a Component's Interfaces

You define the interfaces supported by your component by declaring a set of functions for use by applications. These function declarations specify the parameters that must be provided for each request. The following code illustrates the general form of these function declarations, using the setup request defined for the sample drawing component as an example:

```

FUNCTION DrawerSetup (myInstance: ComponentInstance;
    VAR r: Rect): ComponentResult;

```

This example declares a function that supports the setup request. The first parameter to any component function must be a parameter that accepts a component instance. The Component Manager uses this value to correctly route the request. The calling application must supply a valid component instance when it calls your component. The second and following parameters are those required by your component function. For example, the `DrawerSetup` function takes one additional parameter, a rectangle. Finally, all component functions must return a function result of type `ComponentResult` (a long integer).

Component Manager

These function declarations must also include inline code. This code identifies the request code assigned to the function, specifies the number of bytes of parameter data accepted by the function, and executes a trap to the Component Manager. To continue with the Pascal example used earlier, the inline code for the `DrawerSetup` function is

```
INLINE $2F3C, $0004, $0000, $7000, $A82A;
```

The first element of this code, `$2F3C`, is the opcode for a move instruction that loads the contents of the next two elements onto the stack. The Component Manager uses these values when it invokes your component.

The second element, `$0004`, defines an integer value that specifies the number of bytes of parameter data required by the function, not including the component instance parameter. In this case, the size of a pointer to the rectangle is specified: 4 bytes.

Note

Note that Pascal calling conventions require that Boolean and 1-byte parameters are passed as 16-bit integer values. ♦

The third element, `$0000`, specifies the request code for this function as an integer value. Each function supported by your component must have a unique request code. Your component uses this request code to identify the application's request. You may define only request code values greater than or equal to 0; negative values are reserved for definition by the Component Manager. Recall from the oval drawing component that the request code for the setup request, `kDrawerSetUpSelect`, has a value of 0.

The fourth element, `$7000`, is the opcode for an instruction that sets the D0 register to 0, which tells the Component Manager to call your component rather than to field the request itself.

The fifth element, `$A82A`, is the opcode for an instruction that executes a trap to the Component Manager.

If you are declaring functions for use by Pascal-language applications, your declarations should take the following form:

```
FUNCTION DrawerSetup (myInstance: ComponentInstance;
                     VAR r: Rect): ComponentResult;
INLINE $2F3C, $0004, $0000, $7000, $A82A;
```

If you are declaring functions for use by C-language applications, your declarations can take the following form:

```
pascal ComponentResult DrawerSetup
    (ComponentInstance myInstance, Rect *r) =
    { 0x2F3C, 0x4, 0x0, 0x7000, 0xA82A };
```

Alternatively, you can define the following statement to replace the inline code:

```
#define ComponentCall (callNum, paramSize)
    { 0x2F3C, paramSize, callNum, 0x7000, 0xA82A }
```

Component Manager

Using this statement results in the following declaration format:

```
pascal ComponentResult DrawerSetup
    (ComponentInstance myInstance, Rect *r) =
    ComponentCall (kDrawerSetUpSelect, 4);
```

When a client application calls your function, the system executes the inline code, which invokes the Component Manager. The Component Manager then formats a component parameters record, locates the storage for the current connection, and invokes your component. The Component Manager provides the component parameters record and a handle to the storage of the current connection to your component as function parameters.

Managing Components

This section discusses the Component Manager routines that help you manage your component. It describes how to register your component and how to allow applications to connect to your component.

Registering a Component

Applications must use the services of the Component Manager to find components that meet their needs. Before an application can find a component, however, that component must be registered with the Component Manager. When you register your component, the Component Manager adds the component to its list of available components.

There are two mechanisms for registering a component with the Component Manager. First, during startup processing, the Component Manager searches the Extensions folder (and all of the folders within the Extensions folder) for files of type 'thng'. If the file contains all the information needed for registration (see “Creating a Component Resource” on page 2-60 for more information on creating a component file), the Component Manager automatically registers the component stored in the file. Components registered in this manner are registered globally; that is, the component is made available to all applications and other clients.

Second, your application (or another application) can register your component. When you register your component in this manner, you can specify whether the component should be made available to all applications (global registration) or only to your application (local registration). Your application can register a component that is in memory or that is stored in a resource. You use the `RegisterComponent` function to register a component that is in memory. You use the `RegisterComponentResource` function to register a component that is stored in a component resource. See “The Component Resource” on page 2-107 for a description of the format and content of component resources. The code in Listing 2-14 demonstrates how an application can use the `RegisterComponent` function to register a component that is in memory.

Listing 2-14 Registering a component

```

VAR
    cd:      ComponentDescription;
    draw:    Component;

    WITH cd DO
    BEGIN
        {initialize the component description record}
        componentType := 'draw';
        componentSubtype := 'oval';
        componentManufacturer := 'appl';
        componentFlags := 0;
        componentFlagsMask := 0;
    END;
    {register the component}
    draw := RegisterComponent(cd, ComponentRoutine(@OvalDrawer),
                             0, NIL, NIL, NIL);

```

The code in Listing 2-14 specifies six parameters to the `RegisterComponent` function. The first three are a component description record, a pointer to the component's entry point, and a value of 0 to indicate that this component should be made available only to this application. A component that is registered locally is visible only within the A5 world of the registering program. The last three parameters are specified as `NIL` to indicate that the component doesn't have a name, an information string, or an icon. See page 2-85 for more information on the `RegisterComponent` function.

When a component is registered and the `cmpWantsRegisterMessage` bit is not set in the `componentFlags` field of the component description record, the Component Manager adds the component to its list of registered components. Whenever a client requests access to or information about a component (for example, by using `OpenDefaultComponent`, `FindNextComponent`, or `GetComponentInfo`), the Component Manager searches its list of registered components.

If a component's `cmpWantsRegisterMessage` bit is set, the Component Manager does not automatically add your component to its list of registered components. Instead, it sends your component a series of three requests: open, register, and close. If your component returns a nonzero value as its function result in response to the register request, your component is not added to the Component Manager's list of registered components. Thus, clients are not able to connect to or get information about your component. You might choose to set the `cmpWantsRegisterMessage` bit if, for example, your application requires specific hardware.

Alternatively, you can let your component be automatically registered. Your component can then check for any specific hardware requirements upon receiving an open request. This lets clients attempt to connect to your component and also lets them get information about your component. However, in most cases, if your component requires specific hardware to operate, you should set the `cmpWantsRegisterMessage` bit and respond to the register request appropriately.

Component Manager

If your component controls a hardware resource, you should register your component once for each hardware resource that is available (rather than registering once and allowing multiple instances of your component). This allows clients to easily determine how many hardware resources are available by using the `FindNextComponent` function. If you register a component multiple times, be sure that you specify a unique name for each registration.

If the feature is available, you can request that the Component Manager provide automatic version control for your component (this feature is available only in version 3 and above of the manager). To request automatic version control, specify the `componentDoAutoVersion` flag in the optional extension to the component resource. If you specify this flag, the Component Manager registers your component only if there is no later version available. If an older version is already registered, the Component Manager unregisters it. If a newer version of the same component is registered after yours, the Component Manager automatically unregisters your component. You can use this automatic version control feature to make sure that the most recent version of your component is registered, regardless of the number of versions that are installed.

Creating a Component Resource

You can create a component resource (a resource of type `'thng'`) in a component file. A component file is a file whose resource fork contains a component resource and other required resources for the component. If you store your component in a component file, either you can allow applications to use the `RegisterComponentResource` function to register your component as needed, or you can automatically register your component at startup by storing your component file in the Extensions folder.

Component Manager

A component file consists of

- a component description record that specifies the characteristics of your component (its type, subtype, manufacturer, and control flags)
- the resource type and resource ID of your component's code resource
- the resource type and resource ID of your component's name string
- the resource type and resource ID of your component's information string
- the resource type and resource ID of your component's icon
- optional information about the component (its version number, additional flags, and resource ID of the component's icon family)
- the actual resources for your component's code, name, information string, and icon

Listing 2-15 shows, in Rez format, a component resource that defines an oval drawing component. This drawing component does not specify optional information (see Figure 2-5 on page 2-113 for the contents of the optional extension to the component resource). For compatibility with early versions of the Component Manager, component resources should be locked.

Listing 2-15 Rez input for a component resource

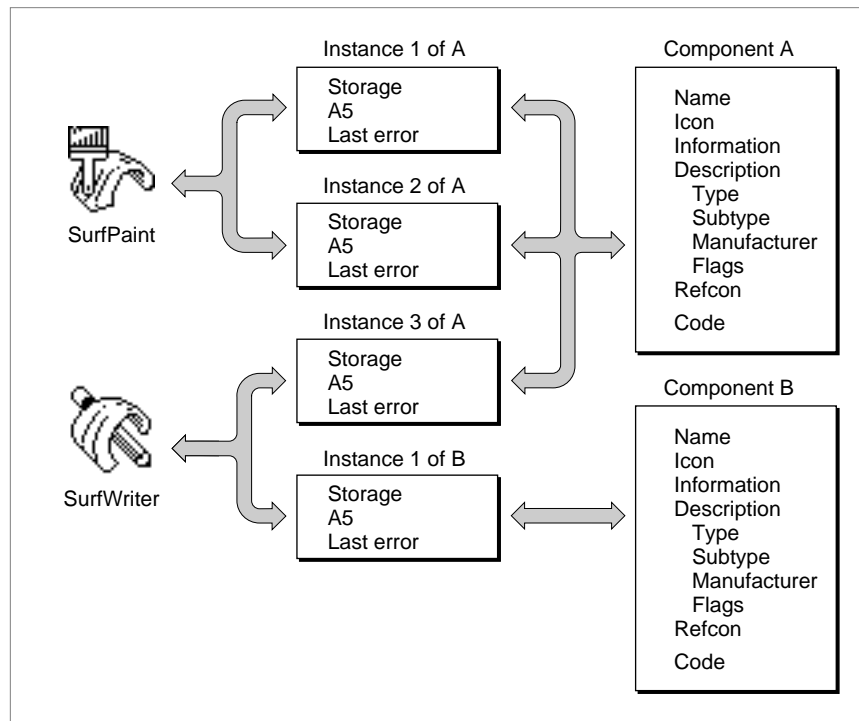
```
resource 'thng' (128, locked) {
    'draw',          /*component type*/
    'oval',          /*component subtype*/
    'appl',          /*component manufacturer*/
    $00000000,       /*component flags: 0*/
    $00000000,       /*reserved (component flags mask): 0*/
    'CODE',          /*component code resource type*/
    128,             /*component code resource ID*/
    'STR ',          /*component name resource type*/
    128,             /*component name resource ID*/
    'STR ',          /*component info resource type*/
    129,             /*component info resource ID*/
    'ICON',          /*component icon resource type*/
    128              /*component icon resource ID*/
    /*optional information (if any) goes here*/
};
```

The component resource, and the resources that define the component's code, name, information string, and icon, must be in the same file. A component file must have the file type 'thng' and reside in the Extensions folder in order to be automatically registered by the Component Manager at startup.

Establishing and Managing Connections

Your component may support one or more connections at a time. In addition, a single application may have open connections with two or more different components at the same time. In fact, a single application can use more than one connection to a single component. Figure 2-2 shows two applications and two components: the first application, SurfPaint, uses two connections to component A; the second application, SurfWriter, uses one connection to component A and one to component B.

Figure 2-2 Supporting multiple component connections



A component can allocate separate storage for each open connection. A component can also set the A5 world for a specific component instance and can maintain separate error information for each instance. A component can also use a reference constant value to maintain global data for the component.

When an application requests that the Component Manager open a connection to your component, the Component Manager issues an open request to your component. At this time, your component should allocate any memory it needs in order to maintain a connection for the requesting application. Be sure to allocate this memory in the current heap zone rather than in the system heap. As described in “Responding to the Open Request” on page 2-47, you can use the `SetComponentInstanceStorage` procedure to associate the allocated memory with the component instance. Whenever the application requests services from your component, the Component Manager supplies you with the

Component Manager

handle to this memory. You can also use the open request as an opportunity to restrict the number of connections your component can support.

To allocate global data for your component, you can maintain a reference constant for use by your component. The Component Manager provides two routines, `SetComponentRefcon` and `GetComponentRefcon`, that allow you to work with your component's reference constant. Note that your component has one reference constant, regardless of the number of connections maintained by your component.

If your component uses its reference constant and is registered globally, be aware that in certain situations the Component Manager may clone your component. This situation occurs only when the Component Manager opens a component that is registered globally and there's no available space in the system heap. In this case, the Component Manager clones your component, creating a new registration of the component in the caller's heap, and returns to the caller the component identifier of the cloned component, not the component identifier of the original registration. The reference constant of the original component is not preserved in the cloned component. Thus you need to take extra steps to set the reference constant of the cloned component to the same value as that of the original component.

To determine whether your component has been cloned, you can examine your component's A5 world using the `GetComponentInstanceA5` function. If the returned value of the A5 world is nonzero, your component is cloned (only components registered globally can be cloned; if your component is registered locally it has a valid, nonzero A5 world and you don't need to check whether it's been cloned). If you determine that your component is cloned, you can retrieve the original reference constant by using the `FindNextComponent` function to iterate through all registrations of your component. You should compare the component identifier of the cloned component with the component identifier returned by `FindNextComponent`. Once you find a component with the same component description but a different component identifier, you've found the original component. You can then use `GetComponentRefcon` to get the reference constant of the original component and then use `SetComponentRefcon` to set the reference constant of the cloned component appropriately. This technique works if a component registers itself only once or registers itself multiple times but with a unique name for each registration. This technique does not work if a component registers itself multiple times using the same name.

When responding to a request from an application, your component can invoke the services of other components. The Component Manager provides two techniques for calling other components. First, your component can call another component using the standard mechanisms also used by applications. The Component Manager then passes the requests to the appropriate component, and your component receives the results of those requests.

Second, your component can redirect a request to another component. For example, you might want to create two similar components that provide different levels of service to applications. Rather than completely implementing both components, you could design one to rely on the capabilities of the other. Use the `DelegateComponentCall` function to pass a request on to another component.

Component Manager

Listing 2-16 shows an example of delegating a request to another component. The component in this example is a drawing component that draws rectangles. The `RectangleDrawer` component handles open, close, and setup requests. It delegates all other requests to another component. When the `RectangleDrawer` component receives an open request, it opens the component to which it will later delegate requests, and stores in its allocated storage the delegated component's component instance. It then specifies this value when it calls the `DelegateComponentCall` function.

Listing 2-16 Delegating a request to another component

```
FUNCTION RectangleDrawer(params: ComponentParameters;
                        storage: Handle): ComponentResult;
VAR
    theRtn: ComponentRoutine;
    safe: Boolean;
BEGIN
    safe := FALSE;
    CASE (params.what) OF
        kComponentOpenSelect:
            theRtn := ComponentRoutine(@RectangleOpen);
        kComponentCloseSelect:
            theRtn := ComponentRoutine(@RectangleClose);
        kDrawerSetupSelect:
            theRtn := ComponentRoutine(@RectangleSetup);
        OTHERWISE
            BEGIN
                safe := TRUE;
                IF (storage <> NIL) THEN
                    RectangleDrawer :=
                        DelegateComponentCall
                            (params,
                             ComponentInstance(StorageHdl(storage)^^.delegateInstance))
                ELSE
                    RectangleDrawer := badComponentSelector;
            END;
    END; {of CASE}
    IF NOT safe THEN
        RectangleDrawer :=
            CallComponentFunctionWithStorage(storage, params, theRtn);
    END;
```


Component Manager Reference

This section provides information about the data structures, routines, and resources defined by the Component Manager. This section is divided into the following topics:

- “Data Structures for Applications” describes the data structures used by applications.
- “Routines for Applications” discusses the Component Manager routines that are available to applications that use components.
- “Data Structures for Components” describes the data structures used by components.
- “Routines for Components” describes the Component Manager routines that are used by components.
- “Application-Defined Routine” describes how to define a component function and supply the appropriate registration information.
- “Resources” describes the format and content of component resources.

Assembly-Language Note

You can invoke Component Manager routines by using the trap `_ComponentDispatch` with the appropriate routine selector. ♦

Data Structures for Applications

This section describes the format and content of the data structures used by applications that use components.

Your application can use the component description record to find components that provide specific services or meet other selection criteria.

The Component Description Record

The component description record identifies the characteristics of a component, including the type of services offered by the component and its manufacturer.

Applications and components use component description records in different ways. An application that uses components specifies the selection criteria for finding a component in a component description record. A component uses the component description record to specify its registration information and capabilities. If you are developing a component, see page 2-80 for information on how a component uses the component description record.

The `ComponentDescription` data type defines the component description record.

Component Manager

```

TYPE  ComponentDescription =
      RECORD
        componentType:      OSType;      {type}
        componentSubType:    OSType;      {subtype}
        componentManufacturer:      {manufacturer}
                                OSType;
        componentFlags:      LongInt;     {control flags}
        componentFlagsMask:  LongInt;     {mask for control }
                                         { flags}
      END;

```

Field descriptions`componentType`

A four-character code that identifies the type of component. All components of a particular type must support a common set of interface routines. For example, drawing components all have a component type of 'draw'.

Your application can use this field to search for components of a given type. You specify the component type in the `componentType` field of the component description record you supply to the `FindNextComponent` or `CountComponents` routine. A value of 0 operates as a wildcard.

`componentSubType`

A four-character code that identifies the subtype of the component. Different subtypes of a component type may support additional features or provide interfaces that extend beyond the standard routines for a given component type. For example, the subtype of drawing components indicates the type of object the component draws. Drawing components that draw ovals have a subtype of 'oval'.

Your application can use the `componentSubType` field to perform a more specific lookup operation than is possible using only the `componentType` field. For example, you may want your application to use only components of a certain component type ('draw') that also have a specific entry in the `componentSubType` field ('oval'). By specifying particular values for both fields in the component description record that you supply to the `FindNextComponent` or `CountComponents` routine, your application retrieves information about only those components that meet both of these search criteria. A value of 0 operates as a wildcard.

Component Manager

`componentManufacturer`

A four-character code that identifies the manufacturer of the component. This field allows for further differentiation between individual components. For example, components made by a specific manufacturer may support an extended feature set. Components provided by Apple use a manufacturer value of 'appl'.

Your application can use this field to find components from a certain manufacturer. Specify the appropriate manufacturer code in the `componentManufacturer` field of the component description record you supply to the `FindNextComponent` or `CountComponents` routine. A value of 0 operates as a wildcard.

`componentFlags`

A 32-bit field that provides additional information about a particular component.

The high-order 8 bits are defined by the Component Manager. You should usually set these bits to 0.

The low-order 24 bits are specific to each component type. These flags can be used to indicate the presence of features or capabilities in a given component.

Your application can use these flags to further narrow the search criteria applied by the `FindNextComponent` or `CountComponents` routine. If you use the `componentFlags` field in a component search, you use the `componentFlagsMask` field to indicate which flags are to be considered in the search.

`componentFlagsMask`

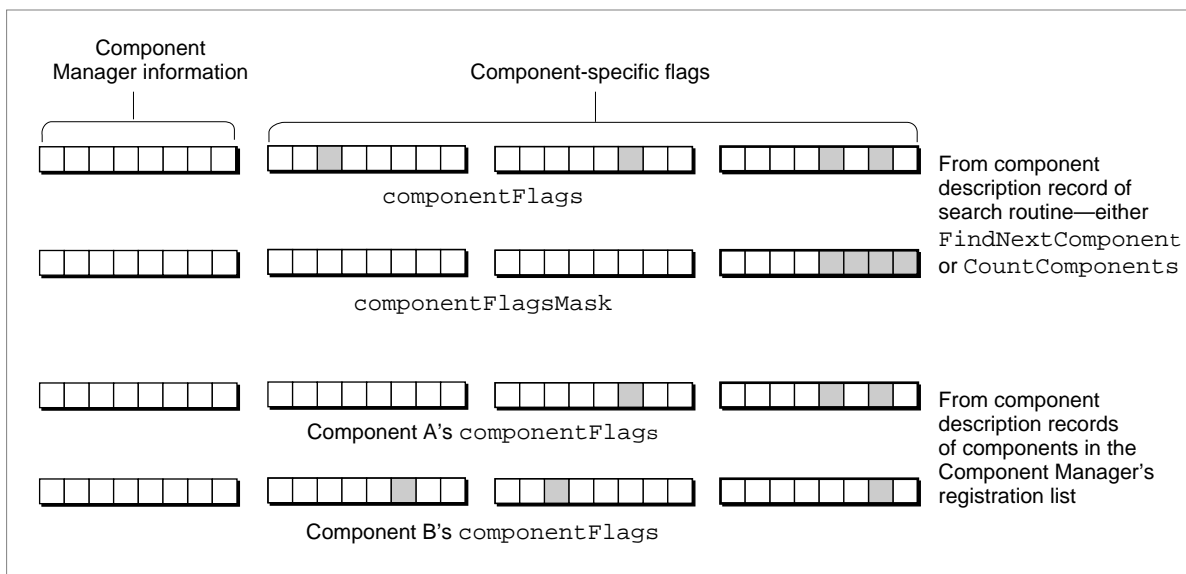
A 32-bit field that indicates which flags in the `componentFlags` field are relevant to a particular component search operation.

For each flag in the `componentFlags` field that is to be considered as a search criterion by the `FindNextComponent` or `CountComponents` routine, your application should set the corresponding bit in the `componentFlagsMask` field to 1. The Component Manager considers only these flags during the search. You specify the desired flag value (either 0 or 1) in the `componentFlags` field.

For example, to look for a component with a specific control flag that is set to 0, set the appropriate bit in the `ComponentFlags` field to 0 and the same bit in the `ComponentFlagsMask` field to 1. To look for a component with a specific control flag that is set to 1, set the bit in the `ComponentFlags` field to 1 and the same bit in the `ComponentFlagsMask` field to 1. To ignore a flag, set the bit in the `ComponentFlagsMask` field to 0.

Figure 2-3 shows how the various fields interact during a search. In the case depicted in the figure, the `componentFlagsMask` field of a component description record supplied to a search routine specifies that only the low-order four flags of the `componentFlags` field are to be examined during the search. The `componentFlags` fields in the component description records of components A and B have a number of flags set. However, in this example the mask specifies that the Component Manager examine only the low-order 4 bits, and therefore only component A meets the search criteria.

Figure 2-3 Interaction between the `componentFlags` and `componentFlagsMask` fields



Note that the `GetComponentInfo` function returns a component identifier in the `componentFlagsMask` field.

Component Identifiers and Component Instances

In general, when using Component Manager routines, your application must specify the particular component using either a component identifier or component instance. The Component Manager identifies *each component* by a component identifier. The Component Manager identifies *each instance* of a component by a component instance. Thus, when your application searches for a component with a particular type and subtype using the `FindNextComponent` function, `FindNextComponent` returns a component identifier that identifies the component. Similarly, your application specifies a component identifier to the `GetComponentInfo` function to obtain information about a component.

Component Manager

When you open a connection to a component, the `OpenDefaultComponent` and `OpenComponent` functions return a component instance. The returned component instance identifies that specific instance of the component. If you open the same component again, the Component Manager returns a different component instance. So a component has a single component identifier and can have multiple component instances. To use a component function, your application specifies a component instance.

Although conceptually component identifiers and component instances serve different purposes, Component Manager routines (with the exception of `DelegateComponentCall`) allow you to use component identifiers and component instances interchangeably. If you do this, you must always coerce the data type appropriately.

A component identifier is defined by the data type `Component`:

```
TYPE
    {component identifier}
    Component          = ^ComponentRecord;
    ComponentRecord    =
    RECORD
        data: ARRAY[0..0] OF LongInt;
    END;
```

A component instance is defined by the data type `ComponentInstance`:

```
TYPE
    {component instance}
    ComponentInstance = ^ComponentInstanceRecord;
    ComponentInstanceRecord =
    RECORD
        data: ARRAY[0..0] OF LongInt;
    END;
```

Routines for Applications

This section discusses the Component Manager routines that are used by applications. If you are developing an application that uses components, you should read this section. If you are developing an application that registers components, you should also read “Registering Components” beginning on page 2-85.

If you are developing a component, you should read this section and “Routines for Components” beginning on page 2-84.

This section describes the routines that allow your application to

- search for components
- gain access to and release components
- get detailed information about specific components

Component Manager

- get component error information

Note

Any of the routines discussed in this section that require a component identifier also accept a component instance. Similarly, you can supply a component identifier to any routine that requires a component instance (except for the `DelegateComponentCall` function). If you do this, you must always coerce the data type appropriately. ♦

Finding Components

The Component Manager provides routines that allow your application to search for components. Your application specifies the search criteria in a component description record. (See “Data Structures for Applications” beginning on page 2-65 for information about the component description record.) Based on the values you specify in fields of the component description record, the Component Manager attempts to find components that meet the needs of your application.

You can use the `CountComponents` function to determine the number of components that match a component description. Use the `FindNextComponent` function to find an individual component that matches a description.

You can use the `GetComponentListModSeed` function to determine whether the list of registered components has changed.

FindNextComponent

The `FindNextComponent` function returns the component identifier for the next registered component that meets the selection criteria specified by your application. You specify the selection criteria in a component description record.

Your application can use the component identifier returned by this function to get more information about the component or to open the component.

```
FUNCTION FindNextComponent (aComponent: Component;
                           looking: ComponentDescription)
                           : Component;
```

`aComponent`

The starting point for the search. Set this field to 0 to start the search at the beginning of the component list. If you are continuing a search, you can specify a component identifier previously returned by the `FindNextComponent` function. The function then searches the remaining components.

`looking`

A component description record. Your application specifies the criteria for the component search in the fields of this record.

The Component Manager ignores fields in the component description record that are set to 0. For example, if you set all the fields to 0, all components meet the search criteria. In this case, your application can retrieve information about all of the components that are registered in the system by repeatedly calling `FindNextComponent` and `GetComponentInfo` until the search is complete. Similarly, if you set all fields to 0 except for the `componentManufacturer` field, the Component Manager searches all registered components for a component supplied by the manufacturer you specify. Note that the `FindNextComponent` function does not modify the contents of the component description record you supply. To retrieve detailed information about a component, you need to use the `GetComponentInfo` function to get the component description record for each returned component.

DESCRIPTION

The `FindNextComponent` function returns the component identifier of a component that meets the search criteria. `FindNextComponent` returns a function result of 0 when there are no more matching components.

SEE ALSO

Use the `GetComponentInfo` function, described on page 2-76, to retrieve more information about a component. To open a component, use the `OpenDefaultComponent` or `OpenComponent` function, described on page 2-73 and page 2-74, respectively. See page 2-65 for information on the component description record.

See Listing 2-1 on page 2-37 for an example of searching for a specific component.

CountComponents

Your application can use the `CountComponents` function to determine the number of registered components that meet your selection criteria. You specify the selection criteria in a component description record. The `CountComponents` function returns the number of components that meet those search criteria.

```
FUNCTION CountComponents (looking: ComponentDescription): LongInt;
```

looking A component description record. Your application specifies the criteria for the component search in the fields of this record.

The Component Manager ignores fields in the component description record that are set to 0. For example, if you set all the fields to 0, the Component Manager returns the number of components registered in the system. Similarly, if you set all fields to 0 except for the

Component Manager

`componentManufacturer` field, the Component Manager returns the number of registered components supplied by the manufacturer you specify.

DESCRIPTION

The `CountComponents` function returns a long integer containing the number of components that meet the specified search criteria.

SEE ALSO

See page 2-65 for information on the component description record.

GetComponentListModSeed

The `GetComponentListModSeed` function allows you to determine if the list of registered components has changed. This function returns the value of the component registration seed number. By comparing this value to values previously returned by the this function, you can determine whether the list has changed. Your application may use this information to rebuild its internal component lists or to trigger other activity that is necessary whenever new components are available.

```
FUNCTION GetComponentListModSeed: LongInt;
```

DESCRIPTION

The `GetComponentListModSeed` function returns a long integer containing the component registration seed number. Each time the Component Manager registers or unregisters a component it generates a new, unique seed number.

Opening and Closing Components

The `OpenDefaultComponent`, `OpenComponent`, and `CloseComponent` functions allow your application to gain access to and release components. Your application must open a component before it can use the services provided by that component. Similarly, your application must close the component when it is finished using the component.

You can use the `OpenDefaultComponent` function to open a component of a specified component type and subtype. You do not have to supply a component description record or call the `FindNextComponent` function to use this function.

You use the `OpenComponent` function to gain access to a specified component. To use this function, your application must have previously obtained a component identifier for the desired component by using the `FindNextComponent` function. (If your application registers a component, it can also obtain a component identifier from the `RegisterComponent` or `RegisterComponentResource` function.)

Once you are finished using a component, use the `CloseComponent` function to release the component.

OpenDefaultComponent

The `OpenDefaultComponent` function allows your application to gain access to the services provided by a component. Your application must open a component before it can call any component functions. You specify the component type and subtype values of the component to open. The Component Manager searches for a component that meets those criteria. If you want to exert more control over the selection process, you can use the `FindNextComponent` and `OpenComponent` functions.

```
FUNCTION OpenDefaultComponent (componentType: OSType;
                               componentSubType: OSType)
                               : ComponentInstance;
```

componentType

A four-character code that identifies the type of component. All components of a particular type support a common set of interface routines. Your application uses this field to search for components of a given type.

componentSubType

A four-character code that identifies the subtype of the component. Different subtypes of a component type may support additional features or provide interfaces that extend beyond the standard routines for a given component type. For example, the subtype of an image compressor component indicates the compression algorithm employed by the compressor.

Your application can use the `componentSubType` field to perform a more specific lookup operation than is possible using only the `componentType` field. For example, you may want your application to use only components of a certain component type ('draw') that also have a specific subtype ('oval'). Set this parameter to 0 to select a component with any subtype value.

DESCRIPTION

The `OpenDefaultComponent` function searches its list of registered components for a component that meets the search criteria. If it finds a component that matches the search criteria, `OpenDefaultComponent` opens a connection to the component and returns a component instance. The returned component instance identifies your application's connection to the component. You must supply this component instance whenever you call the functions provided by the component. When you close the component, you must also supply this component instance to the `CloseComponent` function.

Component Manager

If more than one component in the list of registered components meets the search criteria, `OpenDefaultComponent` opens the first one that it finds in its list.

If it cannot open the specified component, the `OpenDefaultComponent` function returns a function result of `NIL`.

SEE ALSO

For an example that opens a component using the `OpenDefaultComponent` function, see “Opening a Connection to a Default Component” beginning on page 2-35.

OpenComponent

The `OpenComponent` function allows your application to gain access to the services provided by a component. Your application must open a component before it can call any component functions. You specify the component with a component identifier that your application previously obtained from the `FindNextComponent` function.

Alternatively, you can use the `OpenDefaultComponent` function, as previously described, to open a component without calling the `FindNextComponent` function.

Note that your application may maintain several connections to a single component, or it may have connections to several components at the same time.

```
FUNCTION OpenComponent (aComponent: Component): ComponentInstance;
```

`aComponent`

A component identifier that specifies the component to open. Your application obtains this identifier from the `FindNextComponent` function. If your application registers a component, it can also obtain a component identifier from the `RegisterComponent` or `RegisterComponentResource` function.

DESCRIPTION

The `OpenComponent` function returns a component instance. The returned component instance identifies your application’s connection to the component. You must supply this component instance whenever you call the functions provided by the component. When you close the component, you must also supply this component instance to the `CloseComponent` function.

If it cannot open the specified component, the `OpenComponent` function returns a function result of `NIL`.

SEE ALSO

For examples of opening a specific component by using the `FindNextComponent` and `OpenComponent` functions, see Listing 2-1 on page 2-37 and Listing 2-2 on page 2-38, respectively. For a description of the `FindNextComponent` function, see page 2-70.

CloseComponent

The `CloseComponent` function terminates your application's access to the services provided by a component. Your application specifies the connection to be closed with the component instance returned by the `OpenComponent` or `OpenDefaultComponent` function.

```
FUNCTION CloseComponent
    (aComponentInstance: ComponentInstance): OSErr;
```

`aComponentInstance`

A component instance that specifies the connection to close. Your application obtains the component instance from the `OpenComponent` function or the `OpenDefaultComponent` function.

DESCRIPTION

The `CloseComponent` function closes only a single connection. If your application has several connections to a single component, you must call the `CloseComponent` function once for each connection.

RESULT CODES

<code>noErr</code>	0	No error
<code>invalidComponentID</code>	-3000	No component with this component identifier

SEE ALSO

For a description of the `OpenDefaultComponent` and `OpenComponent` functions, see page 2-73 and page 2-74, respectively.

Getting Information About Components

Your application can get the registration information for any component using the `GetComponentInfo` function. You can use the `GetComponentIconSuite` function to get a handle to the component's icon suite, if any.

In addition, for components to which your application already has a connection, your application can obtain the component's version number and also determine whether the component supports a particular request by using the `GetComponentVersion` and `ComponentFunctionImplemented` functions.

GetComponentInfo

The `GetComponentInfo` function returns all of the registration information for a component. Your application specifies the component with a component identifier returned by the `FindNextComponent` function. The `GetComponentInfo` function returns information about the component in a component description record. In addition to the normal component description information, this function returns a component identifier in the `componentFlagsMask` field of the description record.

The `GetComponentInfo` function also returns the component's name, information string, and icon. (To get a handle to the component's icon suite, if it provides one, use the `GetComponentIconSuite` function.)

A component provides this registration information when it is registered with the Component Manager.

```
FUNCTION GetComponentInfo (aComponent: Component;
                           VAR cd: ComponentDescription;
                           componentName: Handle;
                           componentInfo: Handle;
                           componentIcon: Handle): OSErr;
```

`aComponent`

A component identifier that specifies the component for the operation. Your application obtains a component identifier from the `FindNextComponent` function. If your application registers a component, it can also obtain a component identifier from the `RegisterComponent` or `RegisterComponentResource` function.

You may supply a component instance rather than a component identifier to this function. (If you do so, you must coerce the data type appropriately.) Your application can obtain a component instance from the `OpenComponent` function or the `OpenDefaultComponent` function.

`cd`

A component description record. The `GetComponentInfo` function returns information about the specified component in a component description record.

`componentName`

An existing handle that is to receive the component's name. If the component does not have a name, the `GetComponentInfo` function returns an empty handle. Set this field to `NIL` if you do not want to receive the component's name.

`componentInfo`

An existing handle that is to receive the component's information string. If the component does not have an information string, the `GetComponentInfo` function returns an empty handle. Set this field to `NIL` if you do not want to receive the component's information string.

Component Manager

`componentIcon`

An existing handle that is to receive the component's icon. If the component does not have an icon, the `GetComponentInfo` function returns an empty handle. Set this field to `NIL` if you do not want to receive the component's icon.

DESCRIPTION

The `GetComponentInfo` function returns information about the specified component in the `cd`, `componentName`, `componentInfo`, and `componentIcon` parameters.

RESULT CODES

<code>noErr</code>	0	No error
<code>invalidComponentID</code>	-3000	No component with this component identifier

SEE ALSO

For information on the component description record, see page 2-65. For information on the `FindNextComponent` function, see page 2-70. For information on registering components, see “Registering Components” beginning on page 2-85.

For an example of the use of the `GetComponentInfo` function, see Listing 2-3 on page 2-38.

GetComponentIconSuite

The `GetComponentIconSuite` function returns a handle to the component's icon suite (if it provides one).

```
FUNCTION GetComponentIconSuite (aComponent: Component;
                                VAR iconSuite: Handle): OSErr;
```

`aComponent`

A component identifier that specifies the component for the operation. Your application obtains a component identifier from the `FindNextComponent` function. If your application registers a component, it can also obtain a component identifier from the `RegisterComponent` or `RegisterComponentResource` function.

`iconSuite`

`GetComponentIconSuite` returns, in this parameter, a handle to the component's icon suite, if any. If the component has not provided an icon suite, `GetComponentIconSuite` returns `NIL` in this parameter.

DESCRIPTION

The `GetComponentIconSuite` function returns a handle to the component's icon suite. A component provides to the Component Manager the resource ID of its icon family in the optional extensions to the component resource. Your application is responsible for disposing of the returned icon suite handle.

SPECIAL CONSIDERATIONS

The `GetComponentIconSuite` function is available only in version 3 of the Component Manager.

RESULT CODES

<code>noErr</code>	0	No error
<code>invalidComponentID</code>	-3000	No component with this component identifier

SEE ALSO

For information about icon suites and icon families, see the chapter “Icon Utilities” in this book.

GetComponentVersion

The `GetComponentVersion` function returns a component's version number.

```
FUNCTION GetComponentVersion (ci: ComponentInstance): LongInt;
```

<code>ci</code>	The component instance from which you want to retrieve version information. Your application obtains the component instance from the <code>OpenDefaultComponent</code> or <code>OpenComponent</code> function.
-----------------	--

DESCRIPTION

The `GetComponentVersion` function returns a long integer containing the version number of the component you specify. The high-order 16 bits represent the major version, and the low-order 16 bits represent the minor version. The major version specifies the component specification level; the minor version specifies a particular implementation's version number.

ComponentFunctionImplemented

The `ComponentFunctionImplemented` function allows you to determine whether a component supports a specified request. Your application can use this function to determine a component's capabilities.

```
FUNCTION ComponentFunctionImplemented (ci: ComponentInstance;
                                       ftnNumber: Integer)
                                       : LongInt;
```

ci A component instance that specifies the connection for this operation. Your application obtains the component instance from the `OpenDefaultComponent` or `OpenComponent` function.

ftnNumber A request code value. See *Inside Macintosh: QuickTime Components* for information about the request codes supported by the components supplied by Apple with QuickTime. For other components, see the documentation supplied with the component for request code values.

DESCRIPTION

The `ComponentFunctionImplemented` function returns a long integer indicating whether the component supports the specified request. You can interpret this long integer as if it were a Boolean value. If the returned value is `TRUE`, the component supports the specified request. If the returned value is `FALSE`, the component does not support the request.

Retrieving Component Errors

The Component Manager provides a routine that allows your application to retrieve the last error code that was generated by a component instance. Some component routines return error information as their function result. Other component routines set an error code that your application can retrieve using the `GetComponentInstanceError` function. Refer to the documentation supplied with the component for information on how that particular component handles errors.

GetComponentInstanceError

The `GetComponentInstanceError` function returns the last error generated by a specific connection to a component.

```
FUNCTION GetComponentInstanceError
    (aComponentInstance: ComponentInstance): OSErr;
```

Component Manager

`aComponentInstance`

A component instance that specifies the connection from which you want error information. Your application obtains the component instance from the `OpenDefaultComponent` or `OpenComponent` function.

DESCRIPTION

Once you have retrieved an error code, the Component Manager clears the error code for the connection. If you want to retain that error value, you should save it in your application's local storage.

RESULT CODES

<code>noErr</code>	0	No error
<code>invalidComponentID</code>	-3000	No component with this component identifier

Data Structures for Components

This section describes the format and content of the data structures used by components.

Components, and applications that register components, use the component description record to identify a component. A component resource incorporates the information in a component description record and also includes other information. If you are developing a component or an application that registers components, you must be familiar with both the component description record and component resource; see “Resources” beginning on page 2-107 for a description of the component resource.

The Component Manager passes information about a request to your component in a component parameters record.

The Component Description Record

The component description record identifies the characteristics of a component, including the type of services offered by the component and the manufacturer of the component.

Components use component description records to identify themselves to the Component Manager. If your component is stored in a component resource, the information in the component description record must be part of that resource (see the description of the component resource, on page 2-107). If you have developed an application that registers your component, that application must supply a component description record to the `RegisterComponent` function (see “Registering Components” on page 2-85 for information about registering components).

The `ComponentDescription` data type defines the component description record. Note that the valid values of fields in the component description record are determined by the component type specification. For example, all image compressor components must use the `componentSubType` field to specify the compression algorithm used by the compressor.

Component Manager

```

TYPE  ComponentDescription =
      RECORD
        componentType:      OSType;      {type}
        componentSubType:    OSType;      {subtype}
        componentManufacturer:      {manufacturer}
                                   OSType;
        componentFlags:      LongInt;     {control flags}
        componentFlagsMask:  LongInt;     {reserved}
      END;

```

Field descriptions**componentType**

A four-character code that identifies the type of component. All components of a particular type must support a common set of interface routines. For example, drawing components all have a component type of 'draw'.

Your component must support all of the standard routines for the component type specified by this field. Type codes with all lowercase characters are reserved for definition by Apple. See *Inside Macintosh: QuickTime Components* for information about the QuickTime components supplied by Apple. You can define your own component type code as long as you register it with Apple's Component Registry Group.

componentSubType

A four-character code that identifies the subtype of the component. Different subtypes of a component type may support additional features or provide interfaces that extend beyond the standard routines for a given component. For example, the subtype of a drawing component indicates the type of object the component draws. Drawing components that draw ovals have a subtype of 'oval'.

Your component may use this field to indicate more specific information about the capabilities of the component. There are no restrictions on the content you assign to this field. If no additional information is appropriate for your component type, you may set the componentSubType field to 0.

componentManufacturer

A four-character code that identifies the manufacturer of the component. This field allows for further differentiation between individual components. For example, components made by a specific manufacturer may support an extended feature set. Components provided by Apple use a manufacturer value of 'appl'.

Your component uses this field to indicate the manufacturer of the component. You obtain your manufacturer code, which can be the same as your application signature, from Apple's Component Registry Group.

Component Manager

`componentFlags`

A 32-bit field that provides additional information about a particular component.

The high-order 8 bits are reserved for definition by the Component Manager and provide information about the component. The following bits are currently defined:

CONST

```
cmpWantsRegisterMessage = $80000000;
cmpFastDispatch          = $40000000;
```

The setting of the `cmpWantsRegisterMessage` bit determines whether the Component Manager calls this component during registration. Set this bit to 1 if your component should be called when it is registered; otherwise, set this bit to 0. If you want to automatically dispatch requests to your component to the appropriate routine that handles the request (rather than your component calling `CallComponentFunction` or `CallComponentFunctionWithStorage`), set the `cmpFastDispatch` bit. If you set this bit, you must write your component's entry point in assembly language. If you set this bit, the Component Manager calls your component's entry point with the call's parameters, the handle to that instance's storage, and the caller's return address already on the stack. The Component Manager passes the request code in register D0 and passes the stack location of the instance's storage in register A0. Your component can then use the request code in register D0 to directly dispatch the request itself (for example, by using this value as an index into a table of function addresses). Be sure to note that the standard request codes have negative values. Also note that the function parameter that the caller uses to specify the component instance instead contains a handle to the instance's storage. When the component function completes, control returns to the calling application.

For more information about component registration and initialization, see "Responding to the Register Request" on page 2-51.

The low-order 24 bits are specific to each component type. You can use these flags to indicate any special capabilities or features of your component. Your component may use all 24 bits, as appropriate to its component type. You must set all unused bits to 0.

`componentFlagsMask`

Reserved. (However, note that applications can use this field when performing search operations, as described on page 2-67.)

Your component must set the `componentFlagsMask` field in its component description record to 0.

The Component Parameters Record

The Component Manager uses the component parameters record to pass information to your component about a request from an application. The information in this record completely defines the request. Your component services the request as appropriate.

The `ComponentParameters` data type defines the component parameters record.

```
ComponentParameters =
    PACKED RECORD
        flags:      Char;                {reserved}
        paramSize:  Char;                {size of parameters}
        what:       Integer;             {request code}
        params:     ARRAY[0..0] OF LongInt; {actual parameters}
    END;
```

Field descriptions

<code>flags</code>	Reserved for use by Apple.
<code>paramSize</code>	Specifies the number of bytes of parameter data for this request. The actual parameters are stored in the <code>params</code> field.
<code>what</code>	Specifies the type of request. Component designers define the meaning of positive values and assign them to requests that are supported by components of a given type. Negative values are reserved for definition by Apple. Apple has defined these request codes:

```
CONST
    kComponentOpenSelect      = -1; {required}
    kComponentCloseSelect     = -2; {required}
    kComponentCanDoSelect     = -3; {required}
    kComponentVersionSelect   = -4; {required}
    kComponentRegisterSelect  = -5; {optional}
    kComponentTargetSelect    = -6; {optional}
    kComponentUnregisterSelect = -7; {optional}
```

<code>params</code>	An array that contains the parameters specified by the application that called your component. You can use the <code>CallComponentFunction</code> or <code>CallComponentFunctionWithStorage</code> routine to convert this array into a Pascal-style invocation of a subroutine in your component.
---------------------	---

For information on how your component responds to requests, see “Handling Requests for Service” beginning on page 2-46.

Routines for Components

This section describes the Component Manager routines that are used by components. It also discusses routines a component or application can use to register a component. This section first describes the routines for registering components then describes the routines that allow your component to

- extract the parameters from a component parameters record and invoke a subroutine of your component with these parameters
- manage open connections
- associate storage with a specific connection
- pass error information to the Component Manager for later use by the calling application
- store and retrieve your component's reference constant
- open and close its resource file
- call other components
- capture other components
- target a component instance

Note that version 3 and above of the Component Manager supports automatic version control, the unregister request, and icon families. You should test the version number before using any of these features. You can use the `Gestalt` function with the `gestaltComponentMgr` selector to do this. When you specify this selector, `Gestalt` returns in the `response` parameter a 32-bit value indicating the version of the Component Manager that is installed.

If you are developing an application that uses components but does not register them, you do not have to read this material, though it may be interesting to you. For a discussion of the Component Manager routines that support applications that use components, see “Routines for Applications” beginning on page 2-69.

If you are developing an application that registers components, you should read the next section, “Registering Components.” You may also find the other topics in this section interesting.

If you are developing a component, you should read this entire section. For more information about creating components, see “Creating Components” beginning on page 2-41.

Several of the routines discussed in this section use the component parameters record. For a complete description of that structure, see “Data Structures for Components” beginning on page 2-80. For information on the distinction between component identifiers and component instances, see page 2-68.

Note

Any of the routines discussed in this section that require a component identifier also accept a component instance. Similarly, you can supply a component identifier to any routine that requires a component instance (except for the `DelegateComponentCall` function). If you do this, you must always coerce the data type appropriately. For more information, see “Component Identifiers and Component Instances” on page 2-68. ♦

Registering Components

Before a component can be used by an application, the component must be registered with the Component Manager. The Component Manager automatically registers component resources stored in files with file types of 'thng' that are stored in the Extensions folder (for information about the content of component resources, see “Resources” beginning on page 2-107).

Alternatively, you can use either the `RegisterComponent` function or the `RegisterComponentResource` function to register components. Both applications and components can use these routines to register components.

Furthermore, you can use the `RegisterComponentResourceFile` function to register all components specified in a given resource file.

Once you have registered your component, applications can find the component and retrieve information about it using the Component Manager routines described earlier in this chapter in “Routines for Applications” beginning on page 2-69.

Finally, you can use the `UnregisterComponent` function to remove a component from the registration list.

Note

When an application quits, the Component Manager automatically closes any component connections to that application. In addition, if the application has registered components that reside in its heap space, the Component Manager automatically unregisters those components. ♦

RegisterComponent

The `RegisterComponent` function makes a component available for use by applications (or other clients). Once the Component Manager has registered a component, applications can find and open the component using the standard Component Manager routines. To register a component, you provide information identifying the component and its capabilities. The Component Manager returns a component identifier that uniquely identifies the component to the system.

Component Manager

Components you register with the `RegisterComponent` function must be in memory when you call this function. If you want to register a component that is stored in the resource fork of a file, use the `RegisterComponentResource` function. Use the `RegisterComponentResourceFile` function to register all components in the resource fork of a file.

Note that a component residing in your application heap remains registered until your application unregisters it or quits. A component residing in the system heap and registered by your application remains registered until your application unregisters it or until the computer is shut down.

```
FUNCTION RegisterComponent (cd: ComponentDescription;
                           componentEntryPoint: ComponentRoutine;
                           global: Integer;
                           componentName: Handle;
                           componentInfo: Handle;
                           componentIcon: Handle): Component;
```

cd A component description record that describes the component to be registered. You must correctly fill in the fields of this record before calling the `RegisterComponent` function. When applications search for components using the `FindNextComponent` function, the Component Manager compares the attributes you specify here with those specified by the application. If the attributes match, the Component Manager returns the component identifier to the application.

componentEntryPoint The address of the main entry point of the component you are registering. The routine referred to by this parameter receives all requests for the component.

global A set of flags that control the scope of component registration. You can use these flags to specify a value for the `global` parameter:

```
registerCmpGlobal = 1;
```

Specify this flag to indicate that this component should be made available to other applications and clients as well as the one performing the registration. If you do not specify this flag, the component is available for use only by the registering application or component (that is, the component is local to the A5 world of the registering program).

```
registerCmpNoDuplicates = 2;
```

Specify this flag to indicate that if a component with identical characteristics to the one being registered already exists, then the new one should not be registered (`RegisterComponent` returns 0 in this situation). If you do not specify this flag, the component is registered even if a component with identical characteristics to the one being registered already exists.

Component Manager

```
registerCompAfter = 4;
```

Specify this flag to indicate that this component should be registered after all other components with the same component type. Usually components are registered before others with identical descriptions; specifying this flag overrides that behavior.

```
componentName
```

A handle to the component's name. Set this parameter to `NIL` if you do not want to assign a name to the component.

```
componentInfo
```

A handle to the component's information string. Set this parameter to `NIL` if you do not want to assign an information string to the component.

```
componentIcon
```

A handle to the component's icon (a 32-by-32 pixel black-and-white icon). Set this parameter to `NIL` if you do not want to supply an icon for this component. Note that this icon is not used by the Finder; you supply an icon only so that other components or applications can display your component's icon if needed.

DESCRIPTION

The `RegisterComponent` function registers the specified component, recording the information specified in the `cd`, `componentName`, `componentInfo`, and `componentIcon` parameters. The function returns the component identifier assigned to the component by the Component Manager. If it cannot register the component, the `RegisterComponent` function returns a function result of `NIL`.

SEE ALSO

For a complete description of the component description record, see “Data Structures for Components” beginning on page 2-80.

RegisterComponentResource

The `RegisterComponentResource` function makes a component available for use by applications (or other clients). Once the Component Manager has registered a component, applications can find and open the component using the standard Component Manager routines. You provide information identifying the component and specifying its capabilities. The Component Manager returns a component identifier that uniquely identifies the component to the system.

Components you register with the `RegisterComponentResource` function must be stored in a resource file as a component resource (see “The Component Resource” beginning on page 2-107 for a description of the format and content of component resources). If you want to register a component that is in memory, use the `RegisterComponent` function.

Component Manager

The `RegisterComponentResource` function does not actually load the code specified by the component resource into memory. Rather, the Component Manager loads the component code the first time an application opens the component. If the code is not in the same file as the component resource or if the Component Manager cannot find the file, the open request fails.

Note that a component registered locally by your application remains registered until your application unregisters it or quits. A component registered globally by your application remains registered until your application unregisters it or until the computer is shut down.

```
FUNCTION RegisterComponentResource (cr: ComponentResourceHandle;
                                   global: Integer): Component;
```

cr A handle to a component resource that describes the component to be registered. The component resource contains all the information required to register the component.

global A set of flags that controls the scope of component registration. You can use these flags to specify a value for the `global` parameter:

```
registerCmpGlobal = 1;
```

Specify this flag to indicate that this component should be made available to other applications and clients as well as the one performing the registration. If you do not specify this flag, the component is available for use only by the registering application or component (that is, the component is local to the A5 world of the registering program).

```
registerCmpNoDuplicates = 2;
```

Specify this flag to indicate that if a component with identical characteristics to the one being registered already exists, then the new one should not be registered (`RegisterComponentResource` returns 0 in this situation). If you do not specify this flag, the component is registered even if a component with identical characteristics to the one being registered already exists.

```
registerCompAfter = 4;
```

Specify this flag to indicate that this component should be registered after all other components with the same component type. Usually components are registered before others with identical descriptions; specifying this flag overrides that behavior.

DESCRIPTION

The `RegisterComponentResource` function returns the component identifier assigned to the component by the Component Manager. If the `RegisterComponentResource` function could not register the component, it returns a function result of `NIL`.

SEE ALSO

For a description of the format and content of component resources, see “Resources” beginning on page 2-107.

RegisterComponentResourceFile

The `RegisterComponentResourceFile` function registers all component resources in the given resource file according to the flags specified in the `global` parameter.

```
FUNCTION RegisterComponentResourceFile (resRefNum: integer;
                                       global: integer): LongInt;
```

resRefNum The reference number of the resource file containing the components to register.

global A set of flags that control the scope of the registration of the components in the resource file specified in the `resRefNum` parameter. You can use these flags to specify a value for the `global` parameter:

```
registerCmpGlobal = 1;
```

Specify this flag to indicate that each component in the resource file should be made available to other applications and clients as well as the one performing the registration. If you do not specify this flag, each component is available for use only by the registering application or component (that is, the component is local to the A5 world of the registering program).

```
registerCmpNoDuplicates = 2;
```

Specify this flag to indicate that if a component with identical characteristics to the one being registered already exists, then the new one should not be registered (`RegisterComponentResourceFile` returns 0 in this situation). If you do not specify this flag, the component is registered even if a component with identical characteristics to the one being registered already exists.

```
registerCompAfter = 4;
```

Specify this flag to indicate that as `RegisterComponentResourceFile` registers a component, it should register the component after all other components with the same component type. Usually components are registered before others with identical descriptions; specifying this flag overrides that behavior.

DESCRIPTION

The `RegisterComponentResourceFile` function registers components in a resource file. If the `RegisterComponentResourceFile` function successfully registers all components in the specified resource file, `RegisterComponentResourceFile` returns

Component Manager

a function result that indicates the number of components registered. If the `RegisterComponentResourceFile` function could not register one or more of the components in the resource file or if the specified file reference number is invalid, it returns a negative function result.

SEE ALSO

For a description of the format and content of component resources, see “Resources” beginning on page 2-107.

UnregisterComponent

The `UnregisterComponent` function removes a component from the Component Manager’s registration list. Most components are registered at startup and remain registered until the computer is shut down. However, you may want to provide some services temporarily. In that case you dispose of the component that provides the temporary service by using this function.

```
FUNCTION UnregisterComponent (aComponent: Component): OSErr;
```

`aComponent`

A component identifier that specifies the component to be removed. Applications that register components may obtain this identifier from the `RegisterComponent` or `RegisterComponentResource` functions.

DESCRIPTION

The `UnregisterComponent` function removes the component with the specified component identifier from the list of available components. The component to be removed from the registration list must not be in use by any applications or components. If there are open connections to the component, the `UnregisterComponent` function returns a negative result code.

RESULT CODES

<code>noErr</code>	0	No error
<code>invalidComponentID</code>	-3000	No component with this component identifier
<code>validInstancesExist</code>	-3001	This component has open connections

SEE ALSO

If you provide a component that supports the unregister request, see “Responding to the Register Request” on page 2-51 for more information.

Dispatching to Component Routines

This section discusses routines that simplify the process of calling subroutines within your component.

When an application requests service from your component, your component receives a component parameters record containing the information for that request. That component parameters record contains the parameters that the application provided when it called your component. Your component can use this record to access the parameters directly. Alternatively, you can use the routines described in this section to extract those parameters and pass them to a subroutine of your component. By taking advantage of these routines, you can simplify the structure of your component code. For more information about the interface between the Component Manager and your component, see “Creating Components” beginning on page 2-41.

Use the `CallComponentFunction` function to call a component subroutine without providing it access to global data for that connection. Use the `CallComponentFunctionWithStorage` function to call a component subroutine and to pass it a handle to the memory that stores the global data for that connection.

CallComponentFunction

The `CallComponentFunction` function invokes a specified function of your component with the parameters originally provided by the application that called your component. You pass these parameters by specifying the same component parameters record passed to your component’s main entry point.

```
FUNCTION CallComponentFunction (params: ComponentParameters;
                               func: ComponentFunction): LongInt;
```

<code>params</code>	The component parameters record that your component received from the Component Manager.
<code>func</code>	The address of the function that is to handle the request. The Component Manager calls the routine referred to by the <code>func</code> parameter as a Pascal function with the parameters that were originally provided by the application. The routine referred to by this parameter must return a function result of type <code>ComponentResult</code> (a long integer) indicating the success or failure of the operation.

DESCRIPTION

`CallComponentFunction` returns the value that is returned by the routine referred to by the `func` parameter. Your component should use this value to set the current error for this connection.

SPECIAL CONSIDERATIONS

If your component subroutine does not need global data, your component should use `CallComponentFunction`. If your component subroutine requires memory in which to store global data for the component, your component must use `CallComponentFunctionWithStorage`, which is described next.

SEE ALSO

For an example that uses `CallComponentFunction`, see Listing 2-5 on page 2-44. You can use the `SetComponentInstanceError` procedure, described on page 2-97, to set the current error.

CallComponentFunctionWithStorage

The `CallComponentFunctionWithStorage` function invokes a specified function of your component with the parameters originally provided by the application that called your component. You pass these parameters by specifying the same component parameters record that was received by your component's main entry point. The `CallComponentFunctionWithStorage` function also provides a handle to the memory associated with the current connection.

```
FUNCTION CallComponentFunctionWithStorage
    (storage: Handle; params: ComponentParameters;
     func: ComponentFunction): LongInt;
```

<code>storage</code>	A handle to the memory associated with the current connection. The Component Manager provides this handle to your component along with the request.
<code>params</code>	The component parameters record that your component received from the Component Manager.
<code>func</code>	The address of the function that is to handle the request. The Component Manager calls the routine referred to by the <code>func</code> parameter as a Pascal function with the parameters that were originally provided by the application. These parameters are preceded by a handle to the memory associated with the current connection. The routine referred to by the <code>func</code> parameter must return a function result of type <code>ComponentResult</code> (a long integer) indicating the success or failure of the operation.

DESCRIPTION

The `CallComponentFunctionWithStorage` function returns the value that is returned by the function referred to by the `func` parameter. Your component should use this value to set the current error for this connection.

SPECIAL CONSIDERATIONS

`CallComponentFunctionWithStorage` takes as a parameter a handle to the memory associated with the connection, so subroutines of a component that don't need global data should use the `CallComponentFunction` routine described in the previous section instead.

If your component subroutine requires a handle to the memory associated with the connection, you must use `CallComponentFunctionWithStorage`. You allocate the memory for a given connection each time your component is opened. You inform the Component Manager that a connection has memory associated with it by calling the `SetComponentInstanceStorage` procedure.

SEE ALSO

For an example that uses `CallComponentFunctionWithStorage`, see Listing 2-5 on page 2-44. Use the `SetComponentInstanceError` procedure, described on page 2-97, to set the current error for a connection. A description of the `SetComponentInstanceStorage` procedure is given next.

Managing Component Connections

The Component Manager provides a number of routines that help your component manage the connections it maintains with its client applications and components.

Use the `SetComponentInstanceStorage` procedure to inform the Component Manager of the memory your component is using to maintain global data for a connection. Whenever the client application issues a request to the connection, the Component Manager provides to your component the handle to the allocated memory for that connection along with the parameters for the request. You can also use the `GetComponentInstanceStorage` function to retrieve a handle to the storage for a connection.

Use the `CountComponentInstances` function to count all the connections that are currently maintained by your component. This routine is similar to the `CountComponents` routine that the Component Manager provides to client applications and components.

Use the `SetComponentInstanceA5` procedure to set the A5 world for a connection. Once you set the A5 world for a connection, the Component Manager automatically switches the contents of the A5 register when your component receives a request for that connection. When your component returns to the Component Manager, the Component Manager restores the A5 register. Your component can use the `GetComponentInstanceA5` function to retrieve the A5 world for a connection.

SetComponentInstanceStorage

When an application or component opens a connection to your component, the Component Manager sends your component an open request. In response to this open request, your component should set up an environment to service the connection. Typically, your component should allocate some memory for the connection. Your component can then use that memory to maintain state information appropriate to the connection.

The `SetComponentInstanceStorage` procedure allows your component to pass a handle to this memory to the Component Manager. The Component Manager then provides this handle to your component each time the client application requests service from this connection.

PROCEDURE `SetComponentInstanceStorage`

```
(aComponentInstance: ComponentInstance; theStorage: Handle);
```

`aComponentInstance`

The connection to associate with the allocated memory. The Component Manager provides a component instance to your component when the connection is opened.

`theStorage`

A handle to the memory that your component has allocated for the connection. Your component must allocate this memory in the current heap. The Component Manager saves this handle and provides it to your component, along with other parameters, in subsequent requests to this connection.

DESCRIPTION

The `SetComponentInstanceStorage` procedure associates the handle passed in the parameter `theStorage` with the connection specified by the `aComponentInstance` parameter. Your component should dispose of any allocated memory for the connection only in response to the close request.

SPECIAL CONSIDERATIONS

Note that whenever an open request fails, the Component Manager always issues the close request. Furthermore, the value stored with `SetComponentInstanceStorage` is always passed to the close request, so it must be valid or `NIL`. If the open request tries to dispose of its allocated memory before returning, it should call `SetComponentInstanceStorage` again with a `NIL` handle to keep the Component Manager from passing an invalid handle to the close request.

SEE ALSO

For an example that allocates memory in response to an open request, see Listing 2-6 on page 2-48.

GetComponentInstanceStorage

The `GetComponentInstanceStorage` function allows your component to retrieve a handle to the memory associated with a connection. Your component tells the Component Manager about this memory by calling the `SetComponentInstanceStorage` procedure. Typically, your component does not need to use this function, because the Component Manager provides this handle to your component each time the client application requests service from this connection.

```
FUNCTION GetComponentInstanceStorage
    (aComponentInstance: ComponentInstance): Handle;
```

`aComponentInstance`

The connection for which to retrieve the associated memory. The Component Manager provides a component instance to your component when the connection is opened.

DESCRIPTION

The `GetComponentInstanceStorage` function returns a handle to the memory associated with the specified connection.

CountComponentInstances

The `CountComponentInstances` function allows you to determine the number of open connections being managed by a specified component. This function can be useful if you want to restrict the number of connections for your component or if your component needs to perform special processing based on the number of open connections.

```
FUNCTION CountComponentInstances (aComponent: Component): LongInt;
```

`aComponent`

The component for which you want a count of open connections. You can use the component instance that your component received in its open request to identify your component.

DESCRIPTION

The `CountComponentInstances` function returns the number of open connections for the specified component.

SetComponentInstanceA5

The `SetComponentInstanceA5` procedure allows your component to set the A5 world for a connection.

```
PROCEDURE SetComponentInstanceA5
    (aComponentInstance: ComponentInstance; theA5: LongInt);
```

`aComponentInstance`

The connection for which to set the A5 world. The Component Manager provides a component instance to your component when the connection is opened.

`theA5`

The value of the A5 register for the connection. The Component Manager sets the A5 register to this value automatically, and it restores the previous A5 value when your component returns to the Component Manager.

DESCRIPTION

The `SetComponentInstanceA5` procedure sets the A5 world for the specified component instance. Once you set the A5 world for a connection, the Component Manager automatically switches the contents of the A5 register when your component receives a request over that connection. When your component returns to the Component Manager, the Component Manager restores your client's A5 value.

If your component has been registered globally and you have not set an A5 value, the A5 register is set to 0. In this case you should set the A5 world of your component instance to your client's A5 value by using `SetComponentInstanceA5`.

In general, your component uses this procedure only if it is registered globally; in this case, it typically calls `SetComponentInstanceA5` when processing the open request for a new connection.

GetComponentInstanceA5

You can use the `GetComponentInstanceA5` function to retrieve the value of the A5 register for a specified connection. Your component sets the A5 register by calling the `SetComponentInstanceA5` function, as previously described. The Component Manager then sets the A5 register for your component each time the client requests service on this connection. If your component has been registered globally and you have not set an A5 value, the A5 register is set to 0. In this case you should use your client's A5 value.

```
FUNCTION GetComponentInstanceA5
    (aComponentInstance: ComponentInstance): LongInt;
```


Component Manager

`aComponentInstance`

The connection for which to retrieve the A5 value. The Component Manager provides a component instance to your component when the connection is opened.

DESCRIPTION

The `GetComponentInstanceA5` function returns the value of the A5 register for the connection.

Setting Component Errors

The Component Manager maintains error state information for all currently active components. In general, your component returns error information in its function result; a nonzero function result indicates an error occurred, and a function result of 0 indicates the request was successful. However, some requests require that your component return other information as its function result. In these cases, your component can use the `SetComponentInstanceError` procedure to report its latest error state to the Component Manager. You can also use this procedure at any time during your component's execution to report an error.

SetComponentInstanceError

Although your component usually returns error information as its function result, your component can choose to use the `SetComponentInstanceError` procedure to pass error information to the Component Manager. The Component Manager uses this error information to set the current error value for the appropriate connection. Applications can then retrieve this error information by calling the `GetComponentInstanceError` function. The documentation for your component should specify how the component indicates errors.

PROCEDURE `SetComponentInstanceError`

`(aComponentInstance: ComponentInstance; theError: OSErr);`

`aComponentInstance`

A component instance that specifies the connection for which to set the error. The Component Manager provides a component instance to your component when the connection is opened. The Component Manager also provides a component instance to your component as the first parameter in the `params` field of the parameters record.

`theError`

The new value for the current error. The Component Manager uses this value to set the current error for the connection specified by the `aComponentInstance` parameter.

DESCRIPTION

The `SetComponentInstanceError` procedure sets the error associated with the specified component instance to the value specified by the parameter `theError`.

SEE ALSO

For a description of the `GetComponentInstanceError` function, see page 2-79.

Working With Component Reference Constants

The Component Manager provides routines that manage access to the reference constants that are associated with components. There is one reference constant for each component, regardless of the number of connections to that component. When your component is registered, the Component Manager sets this reference constant to 0.

The reference constant is a 4-byte value that your component can use in any way you decide. For example, you might use the reference constant to store the address of a data structure that is shared by all connections maintained by your component. You should allocate shared structures in the system heap. Your component should deallocate the structure when its last connection is closed or when it is unregistered.

Use the `SetComponentRefcon` procedure to set the value of the reference constant for your component. Use the `GetComponentRefcon` function to retrieve the value of the reference constant.

SetComponentRefcon

You can use the `SetComponentRefcon` procedure to set the reference constant for your component.

```
PROCEDURE SetComponentRefcon (aComponent: Component;
                              theRefcon: LongInt);
```

`aComponent`

A component identifier that specifies the component whose reference constant you wish to set.

`theRefcon` The reference constant value that you want to set for your component.

DESCRIPTION

The `SetComponentRefcon` procedure sets the value of the reference constant for your component. Your component can later retrieve the reference constant using the `GetComponentRefcon` function, described next.

GetComponentRefcon

The `GetComponentRefcon` function retrieves the value of the reference constant for your component.

```
FUNCTION GetComponentRefcon (aComponent: Component): LongInt;
```

`aComponent`

A component identifier that specifies the component whose reference constant you wish to get.

DESCRIPTION

The `GetComponentRefcon` function returns a long integer containing the reference constant for the specified component.

Accessing a Component's Resource File

If you store your component in a component resource and register your component using the `RegisterComponentResource` function or `RegisterComponentResourceFile` function, or if the Component Manager automatically registers your component, the Component Manager allows your component to gain access to its resource file. You can store read-only data for your component in its resource file. For example, the resource file may contain the color icon for the component, static data needed to initialize private storage, or any other data that may be useful to the component. Note that there is only one resource file associated with a component.

If you store your component in a component resource but register the component with the `RegisterComponent` function, rather than with the `RegisterComponentResource` or `RegisterComponentResourceFile` function, your component cannot access its resource file with the routines described in this section.

The routines described in this section allow your component to gain access to its resource file. These routines provide read-only access to the data in the resource file. If your component opens its resource file, it must close the file before returning to the calling application.

Use the `OpenComponentResFile` function to open your component's resource file. Use the `CloseComponentResFile` function to close the resource file before returning to the calling application.

OpenComponentResFile

The `OpenComponentResFile` function allows your component to gain access to its resource file. This function opens the resource file with read permission and returns a

Component Manager

reference number that your component can use to read data from the file. The Component Manager adds the resource file to the current resource chain. Your component must close the resource file with the `CloseComponentResFile` function before returning to the calling application.

Your component can use `FSpOpenResFile` or equivalent Resource Manager routines to open other resource files, but you must use `OpenComponentResFile` to open your component's resource file.

```
FUNCTION OpenComponentResFile (aComponent: Component): Integer;
```

`aComponent`

A component identifier that specifies the component whose resource file you wish to open. Applications that register components may obtain this identifier from the `RegisterComponentResource` function.

DESCRIPTION

The `OpenComponentResFile` function returns a reference number for the appropriate resource file. This function returns 0 or a negative number if the specified component does not have an associated resource file or if the Component Manager cannot open the resource file.

Note that when working with resources, your component should always first save the current resource file, perform any resource operations, then restore the current resource file to its previous value before returning.

CloseComponentResFile

This function closes the resource file that your component opened previously with the `OpenComponentResFile` function.

```
FUNCTION CloseComponentResFile (refnum: Integer): OSErr;
```

`refnum`

The reference number that identifies the resource file to be closed. Your component obtains this value from the `OpenComponentResFile` function.

DESCRIPTION

The `CloseComponentResFile` function closes the specified resource file. Your component must close any open resource files before returning to the calling application.

RESULT CODES

<code>noErr</code>	0	No error
<code>resFNotFound</code>	-193	Resource file not found

Calling Other Components

The Component Manager provides two techniques that allow a component to call other components. First, your component may invoke the services of another component using the standard mechanisms also used by applications. The Component Manager then passes the requests to the appropriate component, and your component receives the results of those requests.

Second, your component may supplement its capabilities by using the services of another component to directly satisfy application requests. The Component Manager provides the `DelegateComponentCall` function, which allows your component to pass a request to a specified component. For example, you might want to create two similar components that provide different levels of service to applications. Rather than completely implementing both components, you could design one to rely on the capabilities of the other. In this manner, you have to implement only that portion of the more capable component that provides additional services.

DelegateComponentCall

The `DelegateComponentCall` function provides an efficient mechanism for passing on requests to a specified component. Your component must open a connection to the component to which the requests are to be passed. Your component must close that connection when it has finished using the services of the other component.

Note

The `DelegateComponentCall` function does not accept a component identifier in place of a component instance. In addition, your component should never use the `DelegateComponentCall` function with open or close requests from the Component Manager—always use the `OpenComponent` and `CloseComponent` functions to manage connections with other components. ♦

```
FUNCTION DelegateComponentCall
```

```
    (originalParams: ComponentParameters;  
     ci: ComponentInstance): LongInt;
```

```
originalParams
```

The component parameters record provided to your component by the Component Manager.

Component Manager

ci The component instance that is to process the request. The Component Manager provides a component instance to your component when it opens a connection to another component with the `OpenComponent` or `OpenDefaultComponent` function. You must specify a component instance; this function does not accept a component identifier.

DESCRIPTION

The `DelegateComponentCall` function calls the component instance specified by the `ci` parameter, and passes it the specified component parameters record. `DelegateComponentCall` returns a long integer containing the component result returned by the specified component.

SEE ALSO

See “The Component Parameters Record” on page 2-83 for a description of the component parameters record. See page 2-73, page 2-74, and page 2-75, respectively, for information on the `OpenDefaultComponent`, `OpenComponent`, and `CloseComponent` functions.

See Listing 2-16 on page 2-64 for an example of the use of the `DelegateComponentCall` function.

Capturing Components

The Component Manager allows your component to capture another component. When a component is captured, the Component Manager removes the captured component from its list of available components. The `FindNextComponent` function does not return information about captured components. Also, other applications or clients cannot open or access captured components unless they have previously received a component identifier or component instance for the captured component. The routines described in this section allow your component to capture and uncapture other components.

Typically, your component captures another component when you want to override all or some of the features provided by a component or to provide new features. For example, a component called `NewMath` might capture a component called `OldMath`. Suppose the `NewMath` component provides a new function, `DoExponent`. Whenever `NewMath` gets an exponent request, it can handle the request itself. For all other requests, `NewMath` might call the `OldMath` component to perform the request.

After capturing a component, your component might choose to target a particular instance of the captured component. For information on targeting a component instance, see “Responding to the Target Request” beginning on page 2-52 and “Targeting a Component Instance” on page 2-104.

Use the `CaptureComponent` function to capture a component. Use the `UncaptureComponent` function to restore a previously captured component to the search list.

CaptureComponent

The `CaptureComponent` function allows your component to capture another component. In response to this function, the Component Manager removes the specified component from the search list of components. As a result, applications cannot retrieve information about the captured component or gain access to it. Current clients of the captured component are not affected by this function.

```
FUNCTION CaptureComponent (capturedComponent: Component;
                           capturingComponent: Component)
                           : Component;
```

`capturedComponent`

The component identifier of the component to be captured. Your component can obtain this identifier from the `FindNextComponent` function or from the component registration routines.

`capturingComponent`

The component identifier of your component. Note that you can use the component instance (appropriately coerced) that your component received in its open request in this parameter.

DESCRIPTION

The `CaptureComponent` function removes the specified component from the search list of components and returns a new component identifier. Your component can use this new identifier to refer to the captured component. For example, your component can open the captured component by providing this identifier to the `OpenComponent` function. Your component must provide this identifier to the `UncaptureComponent` function to specify the component to be restored to the search list.

If the component specified by the `capturedComponent` parameter is already captured, the `CaptureComponent` function returns a component identifier set to `NIL`.

SEE ALSO

See “Responding to the Target Request” on page 2-52 and “Targeting a Component Instance” on page 2-104 for information about target requests. For information related to the Component Manager’s use of its list of available components, see page 2-70 for details on the `FindNextComponent` function and page 2-73 for details on the `OpenDefaultComponent` function. See “Registering Components” beginning on page 2-85 for details of the component registration routines.

UncaptureComponent

The `UncaptureComponent` function allows your component to uncapture a previously captured component.

```
FUNCTION UncaptureComponent (aComponent: Component): OSErr;
```

`aComponent`

The component identifier of the component to be uncaptured. Your component obtains this identifier from the `CaptureComponent` function.

DESCRIPTION

The `UncaptureComponent` function restores the specified component to the search list of components. Applications can then access the component and retrieve information about the component using Component Manager routines.

RESULT CODES

<code>noErr</code>	0	No error
<code>invalidComponentID</code>	-3000	No component has this component identifier
<code>componentNotCaptured</code>	-3002	This component has not been captured

Targeting a Component Instance

Your component can target a component instance without capturing the component or your component can first capture the component and then target a specific instance of the component. For information on capturing components, see “Capturing Components” beginning on page 2-102. To target a component instance, use the `ComponentSetTarget` function.

ComponentSetTarget

You can use the `ComponentSetTarget` function to call a component’s target request routine (that is, the routine that handles the `kComponentTargetSelect` request code). The target request informs a component that it has been targeted by another component.

You should not target a component instance if the component does not support the target request. Before calling this function, you should issue a can do request to the component instance you want to target to verify that the component supports the target request. If the component supports it, use the `ComponentSetTarget` function to send a target request to the component instance you wish to target. After receiving a target request, the targeted component instance should call the component instance that targeted it

Component Manager

whenever the targeted component instance would normally call one of its defined functions.

```
FUNCTION ComponentSetTarget (ci: ComponentInstance;
                             target: ComponentInstance): LongInt;
```

ci The component instance to which to send a target request (the component that has been targeted).

target The component instance of the component issuing the target request.

DESCRIPTION

The `ComponentSetTarget` function returns a function result of `badComponentSelector` if the targeted component does not support the target request. Otherwise, the `ComponentSetTarget` function returns as its function result the value that the targeted component instance returned in response to the target request.

SEE ALSO

For details on how to handle the target request, see “Responding to the Target Request” on page 2-52.

Changing the Default Search Order

You can use the `SetDefaultComponent` function to change the order in which the list of registered components is searched.

SetDefaultComponent

The `SetDefaultComponent` function allows your component to change the search order for registered components. You specify a component that is to be placed at the front of the search chain, along with control information that governs the reordering operation. The order of the search chain influences which component the Component Manager selects in response to an application's use of the `OpenDefaultComponent` and `FindNextComponent` functions.

```
FUNCTION SetDefaultComponent (aComponent: Component;
                              flags: Integer): OSErr;
```

aComponent A component identifier that specifies the component for this operation.

Component Manager

<code>flags</code>	A value specifying the control information governing the operation. The value of this parameter controls which component description fields the Component Manager examines during the reorder operation. Set the appropriate flags to 1 to define the fields that are examined during the reorder operation. The following flags are defined:
<code>defaultComponentIdentical</code>	The Component Manager places the specified component in front of all other components that have the same component description.
<code>defaultComponentAnyFlags</code>	The Component Manager ignores the value of the <code>componentFlags</code> field during the reorder operation.
<code>defaultComponentAnyManufacturer</code>	The Component Manager ignores the value of the <code>componentManufacturer</code> field during the reorder operation.
<code>defaultComponentAnySubType</code>	The Component Manager ignores the value of the <code>componentSubType</code> field during the reorder operation.

DESCRIPTION

The `SetDefaultComponent` function changes the search order of registered components by moving the specified component to the front of the search chain, according to the value specified in the `flags` parameter.

SPECIAL CONSIDERATIONS

Note that the `SetDefaultComponent` function changes the search order for all applications. As a result, you should use this function carefully.

RESULT CODES

<code>noErr</code>	0	No error
<code>invalidComponentID</code>	-3000	No component has this component identifier

Application-Defined Routine

To provide a component, you define a component function and supply the appropriate registration information. You store your component function in a code resource and typically store your component's registration information as resources in a component file. For additional information on this process, see "Creating Components" beginning on page 2-41.

MyComponent

Here's how to declare a component function named `MyComponent`:

```
FUNCTION MyComponent (params: ComponentParameters;
                     storage: Handle): ComponentResult;
```

<code>params</code>	A component parameters record. The <code>what</code> field of the component parameters record indicates the action your component should perform. The parameters that the client invoked your function with are contained in the <code>params</code> field of the component parameters record. Your component can use the <code>CallComponentFunction</code> or <code>CallComponentFunctionWithStorage</code> routine to extract the parameters from this record.
<code>storage</code>	A handle to any memory that your component has associated with the connection. Typically, upon receiving an open request, your component allocates memory and uses the <code>SetComponentInstanceStorage</code> function to associate the allocated memory with the component connection.

DESCRIPTION

When your component receives a request, it should perform the action specified in the `what` field of the component parameters record. Your component should return a value of type `ComponentResult` (a long integer). If your component does not return error information as its function result, it should indicate errors using the `SetComponentInstanceError` procedure.

SEE ALSO

For information on the component parameters record, see page 2-83. For information on writing a component, see “Creating Components” beginning on page 2-41.

Resources

This section describes the resource you use to define your component. If you are developing a component, you should be familiar with the format and content of a component resource.

The Component Resource

A component resource (a resource of type `'thng'`) stores all of the information about a component in a single file. The component resource contains all the information needed to register a code resource as a component. Information in the component resource tells the Component Manager where to find the code for the component.

Component Manager

If you are developing an application that uses components, you do not need to know about component resources.

If you are developing a component or an application that registers components, you should be familiar with component resources. The Component Manager automatically registers any components that are stored in component files in the Extensions folder. The file type for component files must be set to 'thng'. If you store your component in a component file in the Extensions folder, you do not need to create an application to register the component.

The Component Manager provides routines that register components. The `RegisterComponent` function registers components that are not stored in resource files. The `RegisterComponentResource` and `RegisterComponentResourceFile` functions register components that are stored as component resources in a component file. If you are developing an application that registers components, you should use the routine that is appropriate to the storage format of the component. For more information about how your application can register components, see “Registering Components” beginning on page 2-85.

This section describes the component resource, which must be provided by all components stored in a component file. Applications that register a component using the `RegisterComponent` function must also provide the same information as that contained in a component resource.

IMPORTANT

For compatibility with early versions of the Component Manager, a component resource must be locked. ▲

The `ComponentResource` data type defines the structure of a component resource. (You can also optionally append to the end of this structure the information defined by the `ComponentResourceExtension` data type, as shown in Figure 2-5 on page 2-113.)

```
ComponentResource =
    RECORD
        cd:                                {registration information}
                                     ComponentDescription;
        component: ResourceSpec; {code resource}
        componentName: ResourceSpec; {name string resource}
        componentInfo: ResourceSpec; {info string resource}
        componentIcon: ResourceSpec; {icon resource}
    END;
```

Field descriptions

cd	A component description record that specifies the characteristics of the component. For a complete description of this record, see page 2-80.
component	A resource specification record that specifies the type and ID of the component code resource. The <code>resType</code> field of the resource

Component Manager

	specification record may contain any value. The component's main entry point must be at offset 0 in the resource.
<code>componentName</code>	A resource specification record that specifies the resource type and ID for the name of the component. This is a Pascal string. Typically, the component name is stored in a resource of type 'STR'.
<code>componentInfo</code>	A resource specification record that specifies the resource type and ID for the information string that describes the component. This is a Pascal string. Typically, the information string is stored in a resource of type 'STR'. You might use the information stored in this resource in a Get Info dialog box.
<code>componentIcon</code>	A resource specification record that specifies the resource type and ID for the icon for a component. Component icons are stored as 32-by-32 bit maps. Typically, the icon is stored in a resource of type 'ICON'. Note that this icon is not used by the Finder; you supply an icon only so that other components or applications can display your component's icon in a dialog box if needed.

A resource specification record, defined by the data type `ResourceSpec`, describes the resource type and resource ID of the component's code, name, information string, or icon. The resources specified by the resource specification records must reside in the same resource file as the component resource itself.

```
ResourceSpec =
    RECORD
        resType:      OSType;      {resource type}
        resId:        Integer;     {resource ID}
    END;
```

You can optionally include in your component resource the information defined by the `ComponentResourceExtension` data type:

```
ComponentResourceExtension =
    RECORD
        componentVersion:      LongInt; {version of component}
        componentRegisterFlags: LongInt; {additional flags}
        componentIconFamily:   Integer; {resource ID of icon }
                                   { family}
    END;
```

Field descriptions`componentVersion`

The version number of the component. If you specify the `componentDoAutoVersion` flag in `componentRegisterFlags`, the Component Manager must obtain the version number of your component when your component is registered. Either you can provide a version number in your component's resource, or you can specify a value of 0 for its version number. If you specify 0, the

Component Manager

Component Manager sends your component a version request to get the version number of your component.

`componentRegisterFlags`

A set of flags containing additional registration information. You can use these constants as flags:

```
CONST
```

```
    componentDoAutoVersion          = 1;  
    componentWantsUnregister        = 2;  
    componentAutoVersionIncludeFlags = 4;
```

Component Manager

Specify the `componentDoAutoVersion` flag if you want the Component Manager to resolve conflicts between different versions of the same component. If you specify this flag, the Component Manager registers your component only if there is no later version available. If an older version is already registered, the Component Manager unregisters it. If a newer version of the same component is registered after yours, the Component Manager automatically unregisters your component. You can use this automatic version control feature to make sure that the most recent version of your component is registered, regardless of the number of versions that are installed.

Specify the `componentWantsUnregister` flag if you want your component to receive an unregister request when it is unregistered.

Specify the flag `componentAutoVersionIncludeFlags` if you want the Component Manager to include the `componentFlags` field of the component description record when it searches for identical components in the process of performing automatic version control for your component. If you do not specify this flag, the Component Manager searches only the `componentType`, `componentSubType`, and `componentManufacturer` fields.

When the Component Manager performs automatic version control for your component, it searches for components with identical values in the `componentType`, `componentSubType`, and `componentManufacturer` fields (and optionally, in the `componentFlags` field). If it finds a matching component, it compares version numbers and registers the most recent version of the component. Note that the setting of the `componentAutoVersionIncludeFlags` flag affects automatic version control only and does not affect the search operations performed by `FindNextComponent` and `CountComponents`.

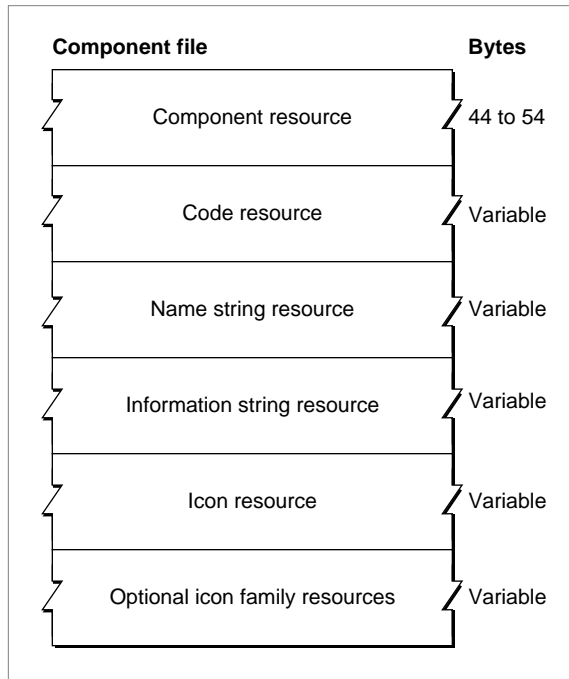
`componentIconFamily`

The resource ID of an icon family. You can provide an icon family in addition to the icon provided in the `componentIcon` field. Note that members of this icon family are not used by the Finder; you supply an icon family only so that other components or applications can display your component's icon in a dialog box if needed.

Component Manager

You store a component resource, along with other resources for the component, in the resource fork of a component file. Figure 2-4 shows the structure of a component file.

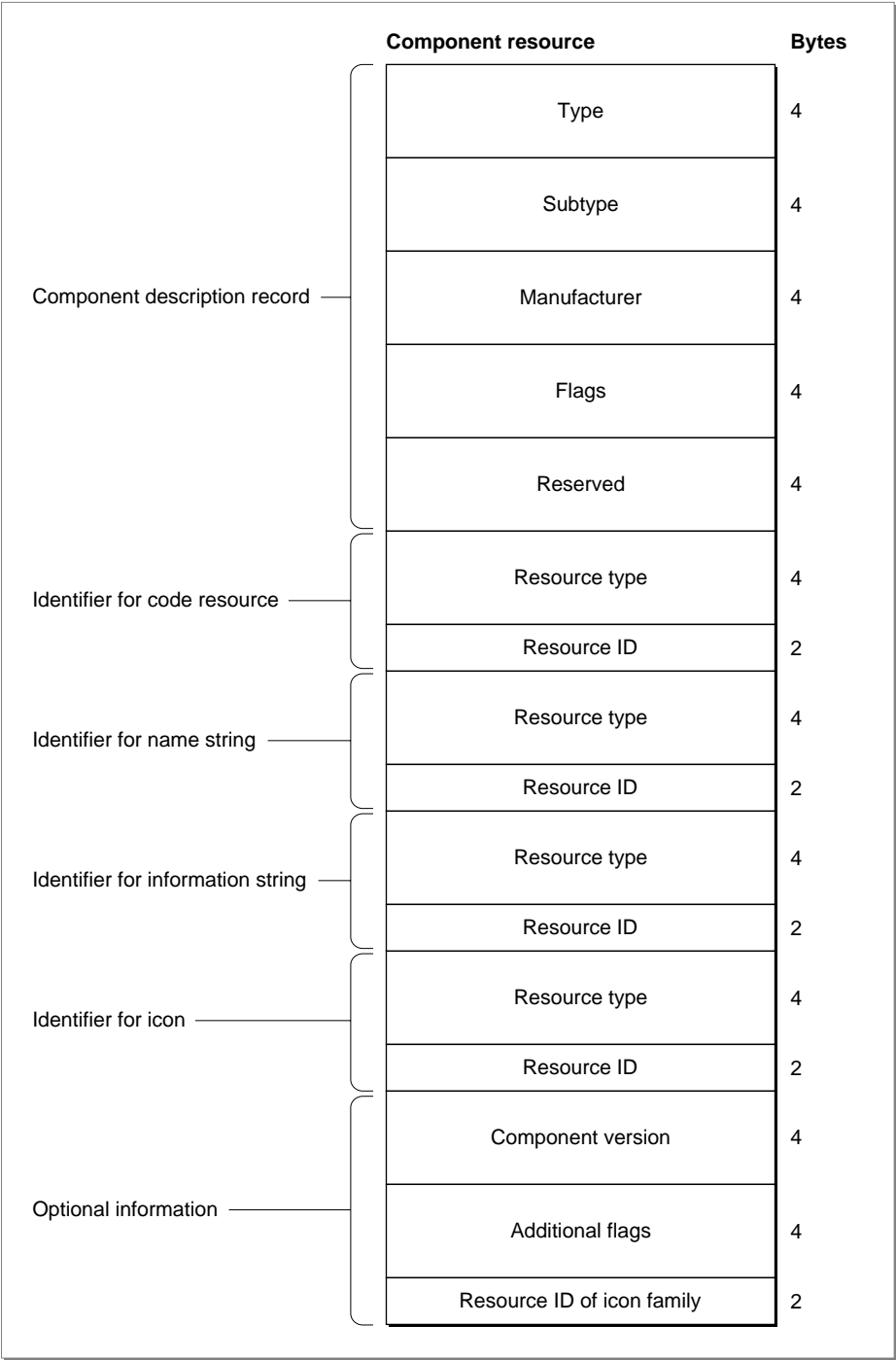
Figure 2-4 Format of a component file



You can also store other resources for your component in your component file. For example, you should include 'FREF', 'BNDL', and icon family resources so that the Finder can associate the icon identifying your component with your component file. When designing the icon for your component file, you should follow the same guidelines as those for system extension icons. See *Macintosh Human Interface Guidelines* for information on designing an icon. See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for information on the 'FREF' and 'BNDL' resources.

Figure 2-5 shows the structure of a component resource.

Figure 2-5 Structure of a compiled component ('thing') resource



Constants

```

#define gestaltComponentMgr 'cpnt'           /*Gestalt selector*/

/*required component routines*/
#define kComponentOpenSelect      -1  /*open request*/
#define kComponentCloseSelect     -2  /*close request*/
#define kComponentCanDoSelect     -3  /*can do request*/
#define kComponentVersionSelect   -4  /*version request*/
#define kComponentRegisterSelect  -5  /*register request*/
#define kComponentTargetSelect    -6  /*target request*/
#define kComponentUnregisterSelect -7  /*unregister request*/

/*wildcard values for searches*/
#define kAnyComponentType         0    /*any type*/
#define kAnyComponentSubType      0    /*any subtype*/
#define kAnyComponentManufacturer 0    /*any manufacturer*/
#define kAnyComponentFlagsMask    0    /*any flags*/

/*component description flags*/
enum {
    cmpWantsRegisterMessage = 1L<<31    /*send register request*/
};

/*flags for optional extension to component resource*/
enum {
    componentDoAutoVersion      = 1,    /*provide version control*/
    componentWantsUnregister     = 2,    /*send unregister request*/
    componentAutoVersionIncludeFlags = 4    /*include flags in search*/
};

enum { /*flags for SetDefaultComponent function*/
    defaultComponentIdentical      = 0,
    defaultComponentAnyFlags       = 1,
    defaultComponentAnyManufacturer = 2,
    defaultComponentAnySubType     = 4,
};

#define defaultComponentAnyFlagsAnyManufacturer
    (defaultComponentAnyFlags+defaultComponentAnyManufacturer)
#define defaultComponentAnyFlagsAnyManufacturerAnySubType
    (defaultComponentAnyFlags+defaultComponentAnyManufacturer
    +defaultComponentAnySubType)

```

Component Manager

```
enum {
/*flags for the global parameter of RegisterComponentResourceFile function*/
    registerCmpGlobal      = 1, /*other apps can communicate with */
                                /* component*/
    registerCmpNoDuplicates = 2, /*duplicate component exists*/
    registerCompAfter      = 4  /*component registered after all others */
                                /* of same type*/
};
```

Result Codes

noErr	0	No error
resFNotFound	-193	Resource file not found
invalidComponentID	-3000	No component has this component identifier
validInstancesExist	-3001	This component has open connections
componentNotCaptured	-3002	This component has not been captured
badComponentInstance	\$800008001	Invalid component passed to Component Manager
badComponentSelector	\$800008002	Component does not support the specified request code

Introduction to File Management

This chapter is a general introduction to file management on Macintosh computers. It explains the basic structure of Macintosh files and the hierarchical file system (HFS) used with Macintosh computers, and it shows how you can use the services provided by the Standard File Package, the File Manager, the Finder, and other system software components to create, open, update, and close files.

You should read this chapter if your application implements the commands typically found in an application's File menu—except for printing commands and the Quit command, which are described elsewhere. This chapter describes how to

- create a new file
- open an existing file
- close a file
- save a document's data in a file
- save a document's data in a file under a new name
- revert to the last saved version of a file
- create and read a preferences file

Depending on the requirements of your application, you may be able to accomplish all your file-related operations by following the instructions given in this chapter. If your application has more specialized file management needs, you'll need to read some or all of the remaining chapters in this book.

This chapter assumes that your application is running in an environment in which the routines that accept file system specification records (defined by the `FSSpec` data type) are available. File system specification records, introduced in system software version 7.0, simplify the identification of objects in the file system. Your development environment may provide "glue" that allows you to call those routines in earlier system software versions. If such glue is not available and you want your application to run in system

software versions earlier than version 7.0, you need to read the discussion of HFS file-manipulation routines in the chapter “File Manager” in this book.

This chapter begins with a description of files and their organization into directories and volumes. Then it describes how to test for the presence of the routines that accept `FSSpec` records and how to use those routines to perform the file management tasks listed above. The chapter ends with descriptions of the data structures and routines used to perform these tasks. The “File Management Reference” and “Summary of File Management” sections in this chapter are subsets of the corresponding sections of the remaining chapters in this book.

About Files

To the user, a file is simply some data stored on a disk. To your application, a **file** is a named, ordered sequence of bytes stored on a Macintosh volume, divided into two forks (as described in the following section, “File Forks”). The information in a file can be used for any of a variety of purposes. For example, a file might contain the text of a letter or the numerical data in a spreadsheet; these types of files are usually known as documents. Typically a **document** is a file that a user can create and edit. A document is usually associated with a single application, which the user expects to be able to open by double-clicking the document’s icon in the Finder.

A file might also contain an application. In that case, the information in the file consists of the executable code of the application itself and any application-specific resources and data. Applications typically allow the user to create and manipulate documents. Some applications also create special files in which they store user-specific settings; such files are known as **preferences files**.

The Macintosh Operating System also uses files for other purposes. For example, the File Manager uses a special file located in a volume to maintain the hierarchical organization of files and folders in that volume. This special file is called the volume’s **catalog file**. Similarly, if virtual memory is in operation, the Operating System stores unused pages of memory in a disk file called the **backing-store file**.

No matter what its function, each file shares certain characteristics with every other file. This section describes these general characteristics of Macintosh files, including

- file forks
- file size and access characteristics
- file system organization
- file naming and identification

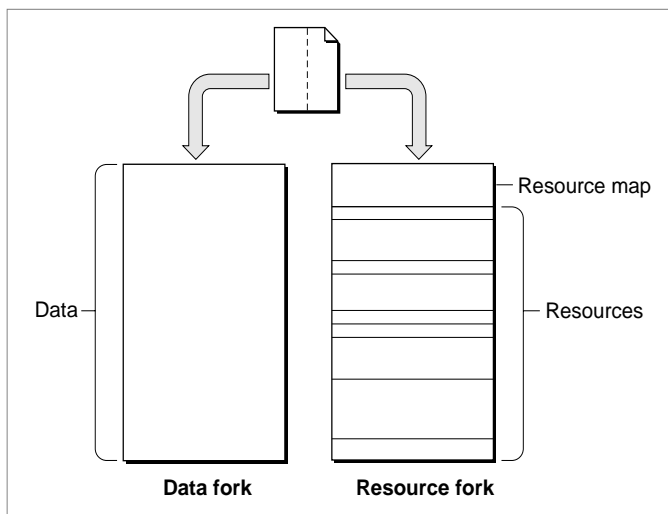
File Forks

Many operating systems treat a file simply as a named, ordered sequence of bytes (possibly terminated by a byte having a special value that indicates the end-of-file). As

illustrated in Figure 3-1, however, each Macintosh file has two **forks**, known as the data fork and the resource fork.

A file's **resource fork** contains that file's resources. If the file is an application, the resource fork typically contains resources that describe the application's menus, dialog boxes, icons, and even the executable code of the application itself. A particularly important resource is the application's 'SIZE' resource, which contains information about the capabilities of the application and its run-time memory requirements. If the file is a document, its resource fork typically contains preference settings, window locations, and document-specific fonts, icons, and so forth.

Figure 3-1 The two forks of a Macintosh file



A file's **data fork** contains the file's data. It is simply a series of consecutive bytes of data. In a sense, the data fork of a Macintosh file corresponds to an entire file in operating systems that treat a file simply as a sequence of bytes. The bytes stored in a file's data fork do not have to exhibit any internal structure, unlike the bytes stored in the resource fork (which consists of a resource map followed by resources). Rather, your application is responsible for interpreting the bytes in the data fork in whatever manner is appropriate. The data fork of a document file might, for example, contain the text of a letter.

Even though a Macintosh file always contains both a resource fork and a data fork, one or both of those forks can be empty. Document files sometimes contain only data (in which case the resource fork is empty). More often, document files contain both resources and data. Application files generally contain resources only (in which case, the data fork is empty). Application files can, however, contain data as well.

Whether you store specific data in the data fork or in the resource fork of a file depends largely on whether that data can usefully be structured as a resource. For example, if you want to store a small number of names and telephone numbers, you can easily define a resource type that pairs each name with its telephone number. Then you can read names

and corresponding numbers from the resource file by using Resource Manager routines. To retrieve the data stored in a resource, you simply specify the resource type and ID; you don't need to know, for instance, how many bytes of data are stored in that resource.

In some cases, however, it is not possible or advisable to store your data in resources. The data might be too difficult to put into the structure required by the Resource Manager. For example, it is easiest to store a document's text, which is usually of variable length, in a file's data fork. Then you can use File Manager routines to access any byte or group of bytes individually.

Even when it is easy to define a resource type for your data, limitations on the Resource Manager might compel you to store your data in the data fork instead. A resource fork can contain at most about 2700 resources. More importantly, the Resource Manager searches linearly through a file's resource types and resource IDs. If the number of types or IDs to be searched is large, accessing the resource data can become slow. As a rule of thumb, if you need to manage data that would occupy more than about 500 resources total, you should use the data fork instead.

IMPORTANT

In general, you should store data created by the user in a file's data fork, unless the data is guaranteed to occupy a small number of resources. The Resource Manager was not designed to be a general-purpose data storage and retrieval system. Also, the Resource Manager does not support multiple access to a file's resource fork. If you want to store data that can be accessed by multiple users of a shared volume, use the data fork. ▲

Because the Resource Manager is of limited use in storing large amounts of user-generated data, most of the techniques in "Using Files" (beginning on page 3-126) illustrate the use of File Manager routines to manage information stored in a file's data fork. See the section "Using a Preferences File" on page 3-150 for an example of the use of the Resource Manager to access data stored in a file's resource fork.

File Size

The size of a file is usually limited only by the size of its volume. A **volume** is a portion of a storage device that is formatted to contain files. A volume can be an entire disk or only a part of a disk. A 3.5-inch floppy disk, for instance, is always formatted as one volume. Other memory devices, such as hard disks and file servers, can contain multiple volumes.

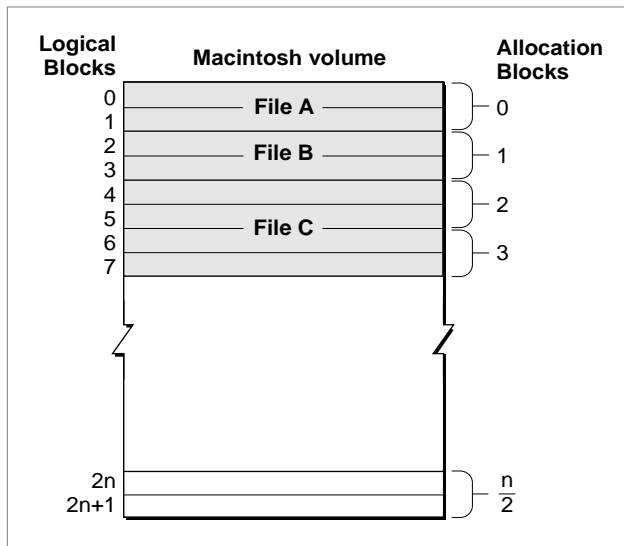
Note

Actually, a file on an HFS volume can be as large as 2 GB (\$7FFFFFFF bytes). Most volumes are not large enough to hold a file of that size. An HFS volume currently can be as large as 2 GB. ♦

The size of a volume varies from one type of device to another. Volumes are formatted into chunks known as **logical blocks**, each of which can contain up to 512 bytes. A double-sided 3.5-inch floppy disk, for instance, usually has 1600 logical blocks, or 800 KB.

Generally, however, the size of a logical block on a volume is of interest only to the disk device driver. This is because the File Manager always allocates space to a file in units called allocation blocks. An **allocation block** is a group of consecutive logical blocks. The File Manager can access a maximum of 65,535 allocation blocks on any volume. For small volumes, such as volumes on floppy disks, the File Manager uses an allocation block size of one logical block. To support volumes larger than about 32 MB, the File Manager needs to use an allocation block size that is at least two logical blocks. To support volumes larger than about 64 MB, the File Manager needs to use an allocation block size that is at least three allocation blocks. In this way, by progressively increasing the number of logical blocks in an allocation block, the File Manager can handle larger and larger volumes. Figure 3-2 illustrates how logical blocks are grouped into allocation blocks.

Figure 3-2 Logical blocks and allocation blocks

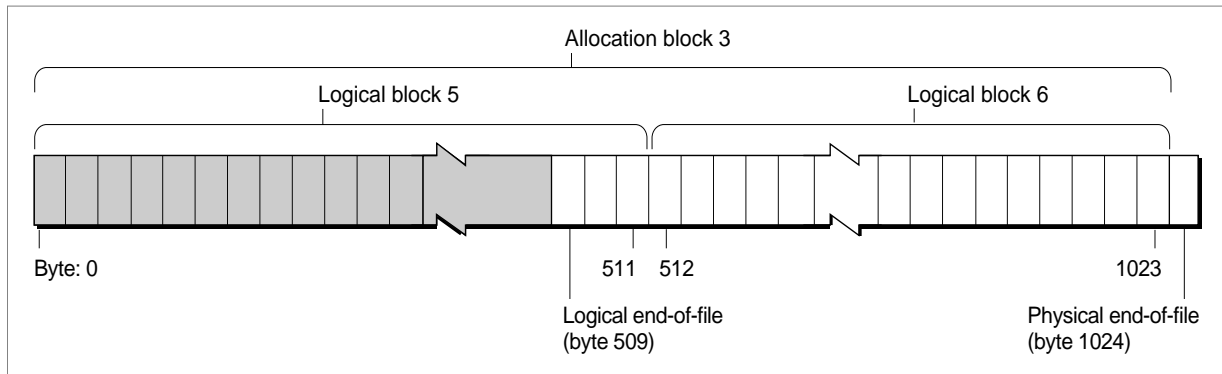


The size of the allocation blocks on a volume is determined when the volume is initialized and depends on the number of logical blocks it contains. In general, the Disk Initialization Manager uses the smallest allocation block size that will allow the File Manager to address the entire volume. A nonempty file fork always occupies at least one allocation block, no matter how many bytes of data that file fork contains. On a 40 MB volume, for example, a file's data fork occupies at least 1024 bytes (that is, two logical blocks), even if it contains only 11 bytes of actual data.

To distinguish between the amount of space allocated to a file and the number of bytes of actual data in the file, two numbers are used to describe the size of a file. The **physical end-of-file** is the number of bytes currently allocated to the file; it's 1 greater than the number of the last byte in its last allocation block (since the first byte is byte number 0). As a result, the physical end-of-file is always an exact multiple of the allocation block size. The **logical end-of-file** is the number of those allocated bytes that currently contain

data; it's 1 greater than the number of the last byte in the file that contains data. For example, on a volume having an allocation block size of two logical blocks (that is, 1024 bytes), a file with 509 bytes of data has a logical end-of-file of 509 and a physical end-of-file of 1024 (see Figure 3-3).

Figure 3-3 Logical end-of-file and physical end-of-file



You can move the logical end-of-file to adjust the size of the file. When you move the logical end-of-file to a position more than one allocation block short of the current physical end-of-file, the File Manager automatically deletes the unneeded allocation block from the file. Similarly, you can increase the size of a file by moving the logical end-of-file past the physical end-of-file. When you move the logical end-of-file past the physical end-of-file, the File Manager automatically adds one or more allocation blocks to the file. The number of allocation blocks added to the file is determined by the volume's clump size. A **clump** is a group of contiguous allocation blocks. The purpose of enlarging files always by adding clumps is to reduce file fragmentation on a volume, thus improving the efficiency of read and write operations.

If you plan to keep extending a file with multiple write operations and you know in advance approximately how large the file is likely to become, you should first call the `SetEOF` function to set the file to that size (instead of having the File Manager adjust the size each time you write past the end-of-file). Doing this reduces file fragmentation and improves I/O performance.

File Access Characteristics

A file can be open or closed. Your application can perform certain operations, such as reading and writing data, only on open files. It can perform other operations, such as deleting, only on closed files.

When you open a file, the File Manager reads information about the file from its volume and stores that information in a **file control block** (FCB). The File Manager also creates an **access path** to the file, a description of the route to be followed when accessing the file. The access path specifies the volume on which the file is located and the location of the

file on the volume. Each access path is assigned a unique **file reference number** (some number greater than 0) by which your application refers to the path. Multiple access paths can be opened to the same file.

For each open access path to a file, the File Manager maintains a current position marker, called the **file mark**, to keep track of where it is in the file during a read or write operation. The mark is the number of the next byte that will be read or written; each time a byte is read or written, the mark is moved. When, during a write operation, the mark reaches the number of the last byte currently allocated to the file, the File Manager adds another clump to the file.

You can read bytes from and write bytes to a file either singly or in sequences of virtually unlimited length. You can specify where each read or write operation should begin by setting the mark or specifying an offset; if you don't, the operation begins at the current file mark.

Each time you want to read or write a file's data, you need to pass the address of a **data buffer**, a part of RAM (usually in your application's heap). The File Manager uses the buffer when it transfers data to or from your application. You can use a single buffer for each read or write operation, or change the address and size of the buffer as necessary.

When your application writes data to a file, the File Manager transfers the data from your application's data buffer and writes it to the **disk cache**, a part of RAM (usually in the System heap). The File Manager uses the disk cache as an intermediate buffer when reading data from or writing it to the file system. When your application requests that data be read from a file, the File Manager looks for the data in the disk cache and transfers it to your application's data buffer if the data is found in the cache; otherwise, the File Manager reads the requested bytes from the disk and puts them in your data buffer.

Note

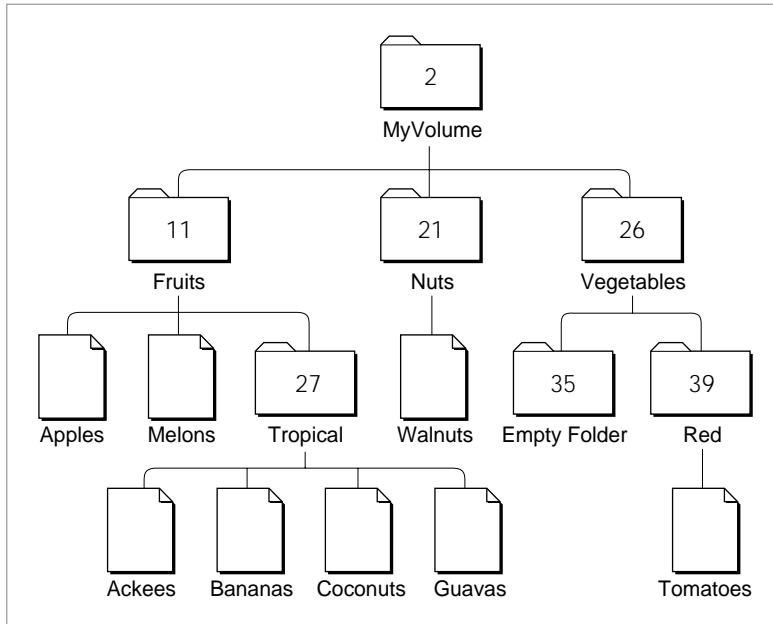
You can also read a continuous stream of characters or a line of characters from a file. In the first case, you ask the File Manager to read a specific number of bytes: When that many have been read, or when the mark reaches the logical end-of-file, the read operation terminates. In the second case, called **newline mode**, the read operation terminates when either of the above conditions is met or when a specified character, the **newline character**, is read. The **newline character** is usually Return (ASCII code \$0D), but it can be any character. Information about newline mode is associated with each access path to a file and can differ from one access path to another. See the chapter "File Manager" in this book for more information about newline mode. ♦

The Hierarchical File System

The Macintosh Operating System uses a method of organizing files called the **hierarchical file system (HFS)**. In HFS, files are grouped into **directories** (also called **folders**), which themselves are grouped into other directories, as illustrated in Figure 3-4. The number listed for each directory is its **directory ID**. The directory ID

is one component of a file system specification, as explained in the next section, “Identifying Files and Directories.”

Figure 3-4 The Macintosh hierarchical file system



The Finder is responsible for managing the files and folders on the desktop. It works with the File Manager to maintain the organization of files and folders on a volume. The hierarchical relationship of folders within folders on the desktop corresponds directly to the hierarchical directory structure maintained on the volume. The volume is known as the **root directory**, and the folders are known as subdirectories, or simply directories.

A volume appears on the desktop only after it has been mounted. Ejectable volumes (such as 3.5-inch floppy disks) are mounted when they’re inserted into a disk drive; nonejectable volumes (such as those on hard disks) are mounted automatically at system startup. When a volume is **mounted**, the File Manager places information about the volume in a nonrelocatable block of memory called a **volume control block (VCB)**. The number of volumes that can be mounted at any time is limited only by the number of drives attached and available memory.

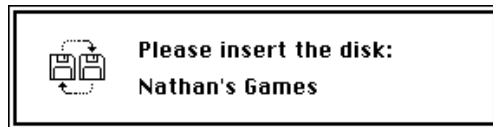
When a volume is mounted, the File Manager assigns a **volume reference number** by which you can refer to the volume for as long as it remains mounted. You can also identify a volume by its **volume name**, a sequence of 1 to 27 printing characters, excluding colons (:). (The File Manager ignores case when comparing names but does recognize diacritical marks.) Whenever possible, though, you should use the volume reference number to avoid confusion between volumes with the same name.

Note

A volume reference number is valid only until the volume is unmounted. If a single volume is mounted and then unmounted, the File Manager may assign it a different volume reference number when it is next mounted. ♦

When an application ejects a 3.5-inch disk from a drive, the File Manager places the volume **offline**. When a volume is offline, the volume control block is kept in memory and the volume reference number is still valid. If you make a File Manager call that specifies that volume, the File Manager presents the disk switch dialog box to the user. Figure 3-5 shows a sample disk switch dialog box.

Figure 3-5 The disk switch dialog box



When the user drags a volume icon to the Trash, that volume is **unmounted**; the volume control block is released, and the volume is no longer known to the File Manager. In particular, the volume reference number previously assigned to the volume is no longer valid.

Each subdirectory is located within a directory called its **parent directory**. Typically, the parent directory is specified by a **parent directory ID**, which is simply the directory ID of the parent directory. The File Manager assigns a special parent directory ID to a volume's root directory. This is primarily to permit a consistent method of identifying files and directories using the volume reference number, the parent directory ID, and the file or directory name. See the next section, "Identifying Files and Directories," for details.

For the most part, your application does not need to be concerned about, or keep track of, the location of files in the file system hierarchy. Most of the files your application opens and saves are specified by the user or another application, and their location is provided to your application by either the Finder or the Standard File Package. One notable exception here concerns preferences files, which are typically stored in the Preferences folder in the currently active System Folder. See "Using a Preferences File" on page 3-150 for instructions on finding preferences files.

Note

In addition to files, folders, and volumes, a fourth type of object, namely an alias, might appear on the Finder desktop. An **alias** is a special kind of file that represents another file, folder, or volume. The Finder and the Standard File Package automatically resolve aliases before passing files to your application, so you generally don't need to do anything with aliases. For more information on working with alias files, see the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* and the chapter "Alias Manager" in this book. ♦

Identifying Files and Directories

The hierarchical arrangement of files and directories allows you to identify a file or directory uniquely by providing just three pieces of information: its volume reference number, its parent directory ID, and its name within that parent directory. The system software lets you specify these three items together in a file system specification record, defined by the `FSSpec` data type:

```

TYPE FSSpec      =          {file system specification}
RECORD
    vRefNum:      Integer;    {volume reference number}
    parID:        LongInt;    {directory ID of parent directory}
    name:         Str63;      {filename or directory name}
END;
```

The `FSSpec` record provides a simple and standard format for specifying files and directories. For example, the Standard File Package procedure `StandardGetFile` uses an `FSSpec` record to return information identifying a user-selected file or folder. You can pass that specification directly to any file-manipulation routines, such as `FSpOpenDF` and `FSpDelete`, that accept `FSSpec` records. In addition, the Alias Manager, Edition Manager, and Finder all use `FSSpec` records to specify files and directories.

Using Files

This section describes how to perform typical file operations using some of the services provided by the Standard File Package, the File Manager, the Finder, and other system software components. Figure 3-6 shows the typical appearance of an application's File menu.

Figure 3-6 A typical File menu

File	
New	⌘N
Open...	⌘O

Close	⌘W
Save	⌘S
Save As...	
Revert to Saved	

Page Setup...	
Print...	⌘P

Quit	⌘Q

Note that all the commands in this menu, except for the Quit and Page Setup commands, manipulate files. Your application's File menu should resemble the menu shown in Figure 3-6 as closely as possible. In general, whenever the user creates or manipulates information that is stored in a document, you need to implement all the commands shown in Figure 3-6.

Note

Some applications allow the user to create or edit information that is not stored in a document. In those cases, it is inappropriate to put the commands that create or manipulate that information in the File menu. Instead, group those commands together in a separate menu. ♦

Listing 3-1 shows one way to handle some of the typical commands in a File menu. Most of the techniques described in this section are illustrated by means of definitions of the functions called in Listing 3-1.

Listing 3-1 Handling the File menu commands

```
PROCEDURE DoHandleFileCommand (menuItem: Integer);
VAR
    myErr: OSErr;
BEGIN
    CASE menuItem OF
        iNew:
            myErr := DoNewCmd;           {create a new document}
        iOpen:
            myErr := DoOpenCmd;          {open an existing document}
        iClose:
            myErr := DoCloseCmd;         {close the current document}
        iSave:
            myErr := DoSaveCmd;          {save the current document}
        iSaveAs:
            myErr := DoSaveAsCmd;        {save document under new name}
        iRevert:
            myErr := DoRevertCmd;        {revert to last saved version}
        OTHERWISE
            ;
    END;
END;
```

Your application should deactivate any menu commands that do not apply to the frontmost window. For example, if the frontmost window is not a document window belonging to your application, then the Close, Save, Save As, and Revert commands should be dimmed when the menu appears. Similarly, if the document in the frontmost window does belong to your application but contains data that has not changed since it

was last saved, then the Save menu command should be dimmed. See “Adjusting the File Menu” on page 3-151 for details on implementing this feature. The definitions of the application-defined functions used in Listing 3-1 assume that this feature has been implemented.

The techniques described in this chapter for manipulating files assume that you identify files and directories by using file system specification records. Because the routines that accept FSSpec records are not available on all versions of system software, you may need to test for the availability of those routines before actually calling any of them. See the next section, “Testing for File Management Routines,” for details.

Testing for File Management Routines

To determine the availability of the routines that operate on FSSpec records, you can call the Gestalt function with the gestaltFSAttr selector code, as illustrated in Listing 3-2.

Listing 3-2 Testing for the availability of routines that operate on FSSpec records

```
FUNCTION FSSpecRoutinesAvail: Boolean;
VAR
    myErr:      OSErr;      {Gestalt result code}
    myFeature:   LongInt;    {Gestalt response}
BEGIN
    FSSpecRoutinesAvail := FALSE;
    IF gHasGestalt THEN      {if Gestalt is available}
        BEGIN
            myErr := Gestalt(gestaltFSAttr, myFeature);
            IF myErr = noErr THEN
                IF BTst(myFeature, gestaltHasFSSpecCalls) THEN
                    FSSpecRoutinesAvail := TRUE;
                END;
            END;
        END;
    END;
```

To use the procedures defined in the following sections to open and save files, you also need to make sure that the routines `StandardGetFile` and `StandardPutFile` are available. You can do this by passing Gestalt the `gestaltStandardFileAttr` selector and verifying that the bit `gestaltStandardFile58` is set in the response value. Also, before using the `FindFolder` function (as shown, for example, in Listing 3-10 on page 3-139), you should call the Gestalt function with the `gestaltFindFolderAttr` selector and verify that the `gestaltFindFolderPresent` bit is set; this indicates that the `FindFolder` function is available.

If the routines that operate on `FSSpec` records are not available, you can use corresponding File Manager and Standard File Package routines. For example, if you cannot call `FSpOpenDF`, you can call `HOpenDF`. That is, instead of writing

```
myErr := FSpOpenDF(mySpec, fsCurPerm, myFile);
```

you can write something like

```
myErr := HOpenDF(myVol, myDirID, myName, fsCurPerm, myFile);
```

The only difference is that the `mySpec` parameter is replaced by three parameters specifying the volume reference number, the parent directory ID, and the filename. With only a few exceptions, all of the techniques presented in this chapter can be easily adapted to work with high-level HFS routines in place of the routines that work with `FSSpec` records.

Note

One notable exception concerns the Standard File Package procedures `SFGetFile` and `SFPutFile`. The `vRefNum` field of the reply record passed to both these functions contains a **working directory reference number**, which encodes both the directory ID and the volume reference number. In general, you should avoid using this number; instead you can turn it into the corresponding directory ID and volume reference number by calling the `GetWDInfo` function. See the chapter “File Manager” in this book for details on working directory reference numbers. ♦

Defining a Document Record

When a user creates a new document or opens an existing document, your application displays the contents of the document in a window, which provides a standard interface for the user to view and possibly edit the document data. It is useful for your application to define a **document record**, an application-specific data structure that contains information about the window, any controls in the window (such as scroll bars), and the file (if any) whose contents are displayed in the window. Listing 3-3 illustrates a sample document record for an application that handles text files.

Listing 3-3 A sample document record

```
TYPE
    MyDocRecHnd    = ^MyDocRecPtr;
    MyDocRecPtr    = ^MyDocRec;
    MyDocRec       =
RECORD
    editRec:       TEHandle;           {handle to TextEdit record}
    vScrollBar:    ControlHandle;      {vertical scroll bar}
```

Introduction to File Management

```

hScrollBar:    ControlHandle;    {horizontal scroll bar}
fileRefNum:    Integer;          {ref num for window's file}
fileFSSpec:    FSSpec;           {file's FSSpec}
windowDirty:   Boolean;          {has window data changed?}
END;
```

Some fields in the `MyDocRec` record hold information about the `TextEdit` record that contains the window's text data. Other fields describe the horizontal and vertical scroll bars in the window. The `MyDocRec` record also contains a field for the file reference number of the open file (if any) whose data is displayed in the window and a field for the file system specification that identifies that file. The file reference number is needed when the application manipulates the open file (for example, when it reads data from or writes data to the file, and when it closes the file). The `FSSpec` record is needed when a "safe-save" procedure is used to save data in a file.

The last field of the `MyDocRec` data type is a Boolean value that indicates whether the contents of the document in the `TextEdit` record differ from the contents of the document in the associated file. When your application first reads a file into the window, you should set this field to `FALSE`. Then, when any subsequent operations alter the contents of the document, you should set the field to `TRUE`. Your application can inspect this field whenever appropriate to determine if special processing is needed. For example, when the user closes a document window and the value of the `windowDirty` flag is `TRUE`, your application should ask the user whether to save the changed version of the document in the file. See Listing 3-16 (page 3-147) for details.

To associate a document record with a particular window, you can simply set a handle to that record as the reference constant of the window (by using the Window Manager procedure `SetWRefCon`). Then you can retrieve the document record by calling the `GetWRefCon` function. Listing 3-15 illustrates this process.

Creating a New File

The user expects to be able to create a new document using the `New` command in the File menu. Listing 3-4 illustrates one way to handle the `New` menu command.

Listing 3-4 Handling the `New` menu command

```

FUNCTION DoNewCmd: OSErr;
VAR
    myWindow:    WindowPtr;    {the new document window; ignored here}
BEGIN
    {Create a new window and make it visible.}
    DoNewCmd := DoNewDocWindow(TRUE, myWindow);
END;
```

The `DoNewCmd` function simply calls the application-defined function `DoNewDocWindow` (shown in Listing 3-6). The first parameter to `DoNewDocWindow` determines whether the new window should be visible or not; the value `TRUE` indicates that the new window should be visible. If `DoNewDocWindow` completes successfully, it returns a window pointer to the calling routine in the second parameter. The `DoNewCmd` function ignores that returned window pointer.

Listing 3-5 Creating a new document window

```

FUNCTION DoNewDocWindow (newDocument: Boolean; var myWindow: WindowPtr):
                                OSErr;

VAR
    myData:      MyDocRecHnd;    {the window's data record}
CONST
    rDocWindow = 1000;           {resource ID of window template}
BEGIN
    {Allocate a new window; see Window Mgr chapter for details.}
    myWindow := GetNewWindow(rDocWindow, NIL, WindowPtr(-1));
    IF myWindow = NIL THEN
        BEGIN
            DoNewDocWindow := MemError;
            Exit(DoNewDocWindow);
        END;

    {Allocate space for the window's data record.}
    myData := MyDocRecHnd(NewHandle(SizeOf(MyDocRec)));
    IF myData = NIL THEN
        BEGIN
            DoNewDocWindow := MemError;
            DisposeWindow(myWindow);
            Exit(DoNewDocWindow);
        END;

    MoveHHi(Handle(myData));           {move the handle high}
    HLock(Handle(myData));             {lock the handle}
    WITH myData^^ DO                  {fill in window data}
        BEGIN
            editRec := TNew(gDestRect, gViewRect);
            vScroll := GetNewControl(rVScroll, myWindow);
            hScroll := GetNewControl(rHScroll, myWindow);
            fileRefNum := 0;            {no file yet!}
            windowDirty := FALSE;
            IF (editRec = NIL) OR (vScroll = NIL) OR (hScroll = NIL) THEN

```

Introduction to File Management

```

BEGIN
    DoNewDocWindow := memFullErr;
    DisposeWindow(myWindow);
    DisposeControl(vScroll);
    DisposeControl(hScroll);
    TEDispose(editRec);
    DisposeHandle(myData);
    Exit(DoNewDocWindow);
END;

END;

IF newDocument THEN                                {if new document, show it}
    ShowWindow(myWindow);

SetWRefCon(myWindow, LongInt(myData)); {link record to window}
HUnlock(Handle(myData));                {unlock the handle}
DoNewDocWindow := noErr;
END;

```

Note that the `DoNewDocWindow` function does not actually create a new file. The reason for this is that it is usually better to wait until the user actually saves a new document before creating a file (mainly because the user might decide not to save the document). The `DoNewDocWindow` function creates a window, allocates a new document record, and fills out the fields of that record. However, it sets the `fileRefNum` field of the document record to 0 to indicate that no file is currently associated with this window.

Opening a File

Your application might need to open a file in several different situations. For example, if the user launches your application by double-clicking one of its document icons in the Finder, the Finder provides your application with information about the selected file (if your application receives high-level events, the Finder sends it an Open Documents event). At that point, you want to create a new window for the document and read the document data from the file into the window.

Your application also opens files after the user chooses the Open command in the File menu. In this case, you need to determine which file to open. You can use the Standard File Package to present a standard dialog box that allows the user to navigate the file system hierarchy (if necessary) and select a file of the appropriate type. Once you get the necessary information from the Standard File Package, you can then create a new window for the document and read the document data from the file into the window.

As you can see, it makes sense to divide the process of opening a document into several different routines. You can have a routine that elicits a file selection from the user and another routine that creates a window and reads the file data into it. In the sample

listings given here, the function `DoOpenCmd` handles the interaction with the user and `DoOpenFile` reads a file into a new window.

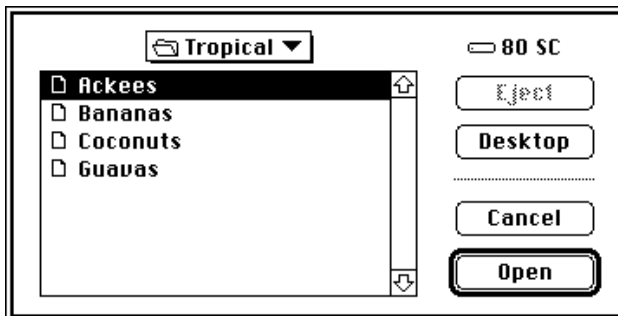
Listing 3-6 shows one way to handle the Open command in the File menu. It uses the Standard File Package routine `StandardGetFile` to determine which file the user wants to open.

Listing 3-6 Handling the Open menu command

```
FUNCTION DoOpenCmd: OSErr;
VAR
    myReply:    StandardFileReply;    {Standard File reply record}
    myTypes:    SFTYPEList;           {types of files to display}
    myErr:      OSErr;
BEGIN
    myErr := noErr;
    myTypes[0] := 'TEXT';              {display text files only}
    StandardGetFile(NIL, 1, myTypes, myReply);
    IF myReply.sfGood THEN
        myErr := DoOpenFile(myReply.sfFile)
    ELSE
        myErr := usrCanceledErr;
    DoOpenCmd := myErr;
END;
```

The `StandardGetFile` procedure requires a list of file types to display in an Open dialog box, as in Figure 3-7. In this case, only text files are to be listed.

Figure 3-7 The default Open dialog box



The user can scroll through the list of files in the current directory, change the current directory, select a file to open, or cancel the operation altogether. When the user clicks either the Cancel or the Open button, `StandardGetFile` fills out the Standard File reply record you pass to it, which has this structure:

```
TYPE StandardFileReply =
  RECORD
    sfGood:      Boolean;      {TRUE if user did not cancel}
    sfReplacing: Boolean;      {TRUE if replacing file with same name}
    sfType:      OSType;      {file type}
    sfFile:      FSSpec;      {selected item}
    sfScript:    ScriptCode; {script of selected item's name}
    sfFlags:     Integer;     {Finder flags of selected item}
    sfIsFolder:  Boolean;     {selected item is a folder}
    sfIsVolume:  Boolean;     {selected item is a volume}
    sfReserved1: LongInt;     {reserved}
    sfReserved2: Integer;     {reserved}
  END;
```

In this situation, the relevant fields of the reply record are the `sfGood` and `sfFile` fields. If the user selects a file to open, the `sfGood` field is set to `TRUE` and the `sfFile` field contains an `FSSpec` record for the selected file. In Listing 3-6, the returned `FSSpec` record is passed directly to the application-defined function `DoOpenFile`. Listing 3-7 illustrates a way to define the `DoOpenFile` function.

Listing 3-7 Opening a file

```
FUNCTION DoOpenFile (mySpec: FSSpec): OSErr;
VAR
  myWindow:      WindowPtr;      {window for file data}
  myData:        MyDocRecHnd;    {handle to window data}
  myFileRefNum:  Integer;        {file reference number}
  myErr:         OSErr;
BEGIN
  {Create a new window, but don't show it yet.}
  myErr := DoNewDocWindow(FALSE, myWindow);
  IF (myErr <> noErr) OR (myWindow = NIL) THEN
    BEGIN
      DoOpenFile := myErr;
      Exit(DoOpenFile);
    END;

  SetWTitle(myWindow, mySpec.name); {set window's title}
  MySetWindowPosition(myWindow);    {set window position}
```

```

{Open the file's data fork for reading and writing.}
myErr := FSpOpenDF(mySpec, fsRdWrPerm, myFileRefNum);
IF myErr <> noErr THEN
    BEGIN
        DisposeWindow(myWindow);
        DoOpenFile := myErr;
        Exit(DoOpenFile);
    END;

{Retrieve handle to window's data record.}
myData := MyDocRecHnd(GetWRefCon(myWindow));
myData^.fileRefNum := myFileRefNum; {save file information}
myData^.fileFSSpec := mySpec;

myErr := DoReadFile(myWindow);           {read in file data}
ShowWindow(myWindow);                    {now show the window}
DoOpenFile := myErr;
END;

```

This function is relatively simple because much of the real work is done by the two functions `DoNewDocWindow` and `DoReadFile`. The `DoReadFile` function is responsible for actually reading the file data from the disk into the `TextEdit` record associated with the document window. See the next section, “Reading File Data,” for a sample definition of `DoReadFile`.

In Listing 3-7, the key step is the call to `FSpOpenDF`, which opens the data fork of the specified file. A file reference number—which indicates an **access path** to the open file—is returned in the third parameter. As you can see, this reference number is saved in the document record, from where it can easily be retrieved for future calls to the `FSRead` and `FSWrite` functions.

The second parameter in a call to the `FSpOpenDF` function specifies the **access mode** for opening the file. For each file, the File Manager maintains access mode information that determines what type of access is available. Most applications support one of two types of access:

- A single user is allowed to read from and write to a file.
- Multiple users are allowed to read from a file, but no one can write to it.

Your application can use the following constants to specify these types of access:

```

CONST
    fsCurPerm      = 0;    {whatever permission is allowed}
    fsRdPerm        = 1;    {read permission}
    fsWrPerm        = 2;    {write permission}
    fsRdWrPerm      = 3;    {exclusive read/write permission}
    fsRdWrShPerm    = 4;    {shared read/write permission}

```

To open a file with exclusive read/write access, you can specify `fsRdWrPerm`. To open a file with read-only access, specify `fsRdPerm`. If you want to open a file and don't know or care which type of access is available, specify `fsCurPerm`. When you specify `fsCurPerm`, if no access paths are already open, the file is opened with exclusive read/write access. If other access paths are already open, but they are read-only, another read-only path is opened.

Reading File Data

Once you have opened a file, you can read data from it by calling the `FSRead` function. Generally you need to read data from a file when the user first opens a file or when the user reverts to the last saved version of a document. The `DoReadFile` function defined in Listing 3-8 illustrates how to use `FSRead` to read data from a file into a `TextEdit` record in either situation.

Listing 3-8 Reading data from a file

```
FUNCTION DoReadFile (myWindow: WindowPtr): OSErr;
VAR
    myData:      MyDocRecHnd;           {handle to a document record}
    myFile:      Integer;               {file reference number}
    myLength:    LongInt;               {number of bytes to read from file}
    myText:      TEHandle;              {handle to TextEdit record}
    myBuffer:    Ptr;                   {pointer to data buffer}
    myErr:       OSErr;
BEGIN
    myData := MyDocRecHnd(GetWRefCon(myWindow)); {get window's data}
    myFile := myData^.fileRefNum;               {get file reference number}
    myErr := SetFPos(myFile, fsFromStart, 0);    {set file mark at start}
    IF myErr <> noErr THEN
        BEGIN
            DoReadFile := myErr;
            Exit(DoReadFile);
        END;

    myErr := GetEOF(myFile, myLength);           {get file length}
    myBuffer := NewPtr(myLength);                 {allocate a buffer}
    IF myBuffer = NIL THEN
        BEGIN
            DoReadFile := MemError;
            Exit(DoReadFile);
        END;
```



```

myErr := FSRead(myFile, myLength, myBuffer); {read data into buffer}
IF (myErr = noErr) OR (myErr = eofErr) THEN
    BEGIN
        {move data into Terec}
        HLock(Handle(myData^^.editRec));
        TereSetText(myBuffer, myLength, myData^^.editRec);
        myErr := noErr;
        HUnlock(Handle(myData^^.editRec));
    END;
DoReadFile := myErr;
END;

```

The `DoReadFile` function takes one parameter specifying the window to read data into. This function first retrieves the handle to that window's document record and extracts the file's reference number from that record. Then `DoReadFile` calls the `SetFPos` function to set the file mark to the beginning of the file having that reference number. There is no need to check that `myFile` has a nonzero value, because `SetFPos` returns an error if you pass it an invalid file reference number.

The second parameter to `SetFPos` specifies the file mark positioning mode; it can contain one of the following values:

```

CONST
    fsAtMark      = 0;  {at current mark}
    fsFromStart   = 1;  {set mark relative to beginning of file}
    fsFromLEOF    = 2;  {set mark relative to logical end-of-file}
    fsFromMark    = 3;  {set mark relative to current mark}

```

If you specify `fsAtMark`, the mark is left wherever it's currently positioned, and the third parameter of `SetFPos` is ignored. The next three constants let you position the mark relative to either the beginning of the file, the logical end-of-file, or the current mark. If you specify one of these three constants, the third parameter contains the byte offset (either positive or negative) from the specified point. Here, the appropriate positioning mode is relative to the beginning of the file.

If `DoReadFile` successfully positions the file mark, it next determines the number of bytes in the file by calling the `GetEOF` function. The key step in the `DoReadFile` function is the call to `FSRead`, which reads the specified number of bytes from the file into the specified buffer. In this case, the data is read into a temporary buffer; then the data is moved into the `TextEdit` record associated with the file. The `FSRead` function returns, in the `myLength` parameter, the number of bytes actually read from the file.

Writing File Data

Generally your application writes data to a file in response to the File menu commands `Save` or `Save As`. However, your application might also incorporate a scheme that automatically saves all open documents to disk every few minutes. It therefore makes sense to isolate the routines that handle the menu commands from the routines that

handle the actual writing of data to disk. This section shows how to write the data stored in a TextEdit record to a file. See “Saving a File” on page 3-140 for instructions on handling the Save and Save As menu commands.

It is very easy to write data from a specified buffer into a specified file. You simply position the file mark at the beginning of the file (using `SetFPos`), write the data into the file (using `FSWrite`), and then resize the file to the number of bytes actually written (using `SetEOF`). Listing 3-9 illustrates this sequence.

Listing 3-9 Writing data into a file

```
FUNCTION DoWriteData (myWindow: WindowPtr; myTemp: Integer): OSErr;
VAR
    myData:      MyDocRecHnd;           {handle to a document record}
    myLength:    LongInt;               {number of bytes to write to file}
    myText:      TEHandle;              {handle to TextEdit record}
    myBuffer:    Handle;                {handle to actual text in Terec}
    myVol:       Integer;               {volume reference number of myFile}
    myErr:       OSErr;
BEGIN
    myData := MyDocRecHnd(GetWRefCon(myWindow)); {get window's data record}
    myText := myData^.editRec;                 {get Terec}
    myBuffer := myText^.hText;                  {get text buffer}
    myLength := myText^.teLength;               {get text buffer size}

    myErr := SetFPos(myTemp, fsFromStart, 0);   {set file mark at start}
    IF myErr = noErr THEN                       {write buffer into file}
        myErr := FSWrite(myTemp, myLength, myBuffer^);
    IF myErr = noErr THEN                       {adjust file size}
        myErr := SetEOF(myTemp, myLength);
    IF myErr = noErr THEN                       {find volume file is on}
        myErr := GetVRefNum(myTemp, myVol);
    IF myErr = noErr THEN                       {flush volume}
        myErr := FlushVol(NIL, myVol);
    IF myErr = noErr THEN                       {show file is up to date}
        myData^.windowDirty := FALSE;
    DoWriteData := myErr;
END;
```

The `DoWriteData` function first retrieves the TextEdit record attached to the specified window and extracts the address and length of the actual text buffer from that record. Then it calls `SetFPos`, `FSWrite`, and `SetEOF` as just explained. Finally, `DoWriteData` determines the volume containing the file (using the `GetVRefNum` function) and flushes that volume (using the `FlushVol` function). This is necessary to ensure that both the file's data and the file's catalog entry are updated.

Notice that the `DoWriteData` function takes a second parameter, `myTemp`, which should be the file reference number of a temporary file, not the file reference number of the file associated with the window whose data you want to write. If you pass the reference number of the file associated with the window, you risk corrupting the file, because the existing file data is overwritten when you position the file mark at the beginning of the file and call `FSWrite`. If `FSWrite` does not complete successfully, it is very likely that the file on disk does not contain the correct document data.

To avoid corrupting the file containing the saved version of a document, always call `DoWriteData` specifying the file reference number of some new, temporary file. Then, when `DoWriteData` completes successfully, you can call the `FSExchangeFiles` function to swap the contents of the temporary file and the existing file. Listing 3-10 illustrates how to update a file on disk safely; it shows a sequence of updating, renaming, saving, and deleting files that preserves the contents of the existing file until the new version is safely recorded.

Listing 3-10 Updating a file safely

```
FUNCTION DoWriteFile (myWindow): OSErr;
VAR
    myData:      MyDocRecHnd;    {handle to window's document record}
    myFSpec:     FSSpec;          {FSSpec for file to update}
    myTSpec:     FSSpec;          {FSSpec for temporary file}
    myTime:      LongInt;         {current time; for temporary filename}
    myName:      Str255;          {name of temporary file}
    myTemp:      Integer;         {file reference number of temporary file}
    myVRef:      Integer;         {volume reference number of temporary file}
    myDirID:     LongInt;         {directory ID of temporary file}
    myErr:       OSErr;
BEGIN
    myData := MyDocRecHnd(GetWRefCon(myWindow)); {get that window's data}
    myFSpec := myData^.fileFSSpec;               {get FSSpec for existing file}

    GetDateTime(myTime);                          {create a temporary filename}
    NumToString(myTime, myName);

    {Find the temporary folder on file's volume; create it if necessary.}
    myErr := FindFolder(myFSpec.vRefNum, kTemporaryFolderType,
                        kCreateFolder, myVRef, myDirID);
    IF myErr = noErr THEN                          {make an FSSpec for temp file}
        myErr := FSMakeFSSpec(myVRef, myDirID, myName, myTSpec);
    IF (myErr = noErr) OR (myErr = fnfErr) THEN {create a temporary file}
        myErr := FSpCreate(myTSpec, 'trsh', 'trsh', smSystemScript);
    IF myErr = noErr THEN                          {open the newly created file}
```

Introduction to File Management

```

    myErr := FSpOpenDF(myTSpec, fsRdWrPerm, myTemp);
IF myErr = noErr THEN                                {write data to the data fork}
    myErr := DoWriteData(myWindow, myTemp);
IF myErr = noErr THEN                                {close the temporary file}
    myErr := FSClose(myTemp);
IF myErr = noErr THEN                                {swap data in the two files}
    myErr := FSpExchangeFiles(myTSpec, myFSpec);
IF myErr = noErr THEN                                {delete the temporary file}
    myErr := FSpDelete(myTSpec);
DoWriteFile := myErr;
END;

```

The essential idea behind this “safe-save” process is to save the data in memory into a new file and then to exchange the contents of the new file and the old version of the file by calling `FSpExchangeFiles`. The `FSpExchangeFiles` function does not move the data on the volume; it merely changes the information in the volume’s catalog file and, if the files are open, in their file control blocks (FCBs). The catalog entry for a file contains

- fields that describe the physical data, such as the first allocation block, physical end, and logical end of both the resource and data forks
- fields that describe the file within the file system, such as file ID and parent directory ID

Fields that describe the data remain with the data; fields that describe the file remain with the file. The creation date remains with the file; the modification date remains with the data. (For a more complete description of the `FSpExchangeFiles` function, see the chapter “File Manager” in this book.)

Saving a File

There are several ways for a user to indicate that the current contents of a document should be saved (that is, written to disk). The user can choose the File menu commands Save or Save As, or the user can click the Save button in a dialog box that you display when the user attempts to close a “dirty” document (that is, a document whose contents have changed since the last time it was saved). You can handle the Save menu command quite easily, as illustrated in Listing 3-11.

Listing 3-11 Handling the Save menu command

```

FUNCTION DoSaveCmd: OSErr;
VAR
    myWindow:    WindowPtr;           {pointer to the front window}
    myData:      MyDocRecHnd;         {handle to a document record}
    myErr:       OSErr;

```

```

BEGIN
    myWindow := FrontWindow;           {get front window and its data}
    myData := MyDocRecHnd(GetWRefCon(myWindow));
    IF myData^.fileRefNum <> 0 THEN      {if window has a file already}
        myErr := DoWriteFile(myWindow); {then write contents to disk}
    ELSE
        myErr := DoSaveAsCmd;           {else ask for a filename}
    DoSaveCmd := myErr;
END;

```

The DoSaveCmd function simply checks whether the frontmost window is already associated with a file. If so, then DoSaveCmd calls DoWriteFile to write the data to disk (using the “safe-save” process illustrated in the previous section). Otherwise, if no file exists for that window, DoSaveCmd calls DoSaveAsCmd. Listing 3-12 shows a way to define the DoSaveAsCmd function.

Listing 3-12 Handling the Save As menu command

```

FUNCTION DoSaveAsCmd: OSErr;
VAR
    myWindow:   WindowPtr;           {pointer to the front window}
    myData:     MyDocRecHnd;          {handle to a document record}
    myReply:    StandardFileReply;
    myFile:     Integer;              {file reference number}
    myErr:      OSErr;
BEGIN
    myWindow := FrontWindow;          {get front window and its data}
    myData := MyDocRecHnd(GetWRefCon(myWindow));
    myErr := noErr;

    StandardPutFile('Save as:', 'Untitled', myReply);
    IF myReply.sfGood THEN              {user saves file}
        BEGIN
            IF NOT myReply.sfReplacing THEN
                myErr := FSpCreate(myReply.sfFile, 'MYAP', 'TEXT',
                                   smSystemScript);

            IF myErr <> noErr THEN
                Exit(DoSaveAsCmd);
            myData^.fileFSSpec := myReply.sfFile;

            IF myData^.fileRefNum <> 0 THEN      {if window already has a file}
                myErr := FSClose(myData^.fileRefNum); {close it}
        END
    END;

```

Introduction to File Management

```

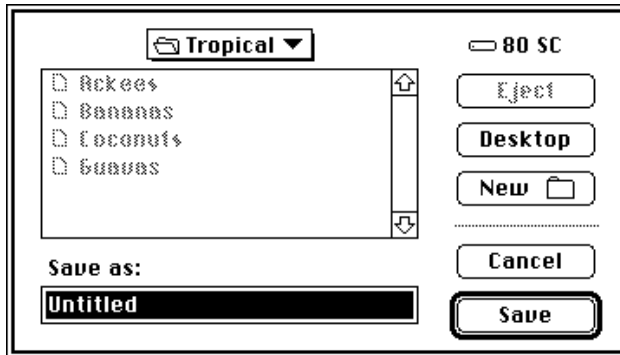
{Create document's resource fork and copy Finder resources to it.}
FSpCreateResFile(myData^.fileFSSpec, 'MYAP', 'TEXT',
                 smSystemScript);
myErr := ResError;
IF myErr = noErr THEN
    myFile := FSpOpenResFile(myData^.fileFSSpec, fsRdWrPerm);
IF myFile > 0 THEN
    {copy Finder resources}
    myErr := DoCopyResource('STR ', -16396, gAppsResFile, myFile)
ELSE
    myErr := ResError;
IF myErr = noErr THEN
    myErr := FSClose(myFile);          {close the resource fork}

{Open data fork and leave it open.}
IF myErr = noErr THEN
    myErr := FSpOpenDF(myData^.fileFSSpec, fsRdWrPerm, myFile);
IF myErr = noErr THEN
    BEGIN
        myData^.fileRefNum := myFile;
        SetWTitle(myWindow, myReply.sfFile.name);
        myErr := DoWriteFile(myWindow);
    END;
DoSaveAsCmd := myErr;
END;
END;

```

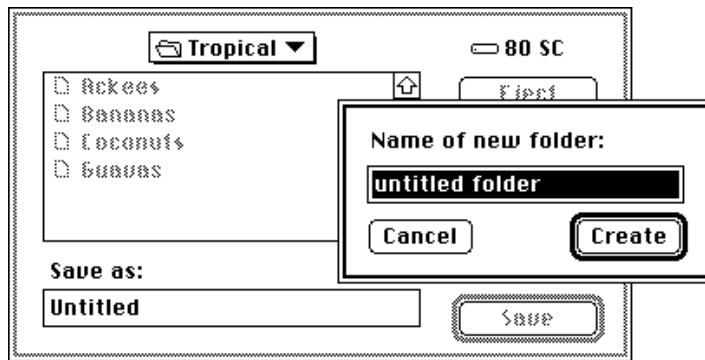
The `StandardPutFile` procedure is similar to the `StandardGetFile` procedure discussed earlier in this chapter. It manages the user interface for the default Save dialog box, illustrated in Figure 3-8.

Figure 3-8 The default Save dialog box



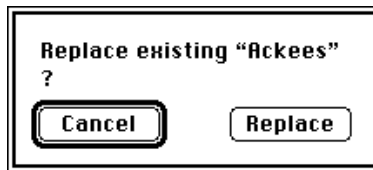
If the user clicks the New Folder button, the Standard File Package presents a subsidiary dialog box like the one shown in Figure 3-9.

Figure 3-9 The new folder dialog box



If the user asks to save a file with a name that already exists at the specified location, the Standard File Package displays a subsidiary dialog box, like the one shown in Figure 3-10, to verify that the new file should replace the existing file.

Figure 3-10 The name conflict dialog box



Note in Listing 3-12 that if the user is not replacing an existing file, the `DoSaveAsCmd` function creates a new file and records the new `FSSpec` record in the window's document record. Otherwise, if the user is replacing an existing file, `DoSaveAsCmd` simply records, in the window's document record, the `FSSpec` record returned by `StandardGetFile`.

When `DoSaveAsCmd` creates a new file, it also copies a resource from your application's resource fork to the resource fork of the newly created file. This resource (with ID -16396) identifies the name of your application. (For more details about this resource, see the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials*.) The `DoSaveAsCmd` function calls the application-defined routine `DoCopyResource`. Listing 3-13 shows a simple way to define the `DoCopyResource` function.

Listing 3-13 Copying a resource from one resource fork to another

```

FUNCTION DoCopyResource (theType: ResType; theID: Integer;
                        source: Integer; dest: Integer): OSErr;

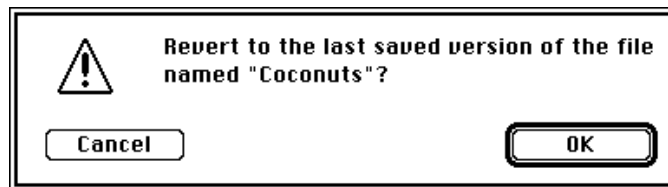
VAR
    myHandle:   Handle;           {handle to resource to copy}
    myName:     Str255;           {name of resource to copy}
    myType:     ResType;          {ignored; used for GetResInfo}
    myID:       Integer;          {ignored; used for GetResInfo}
BEGIN
    UseResFile(source);           {set the source resource file}
    myHandle := GetResource(theType, theID); {open the source resource}
    IF myHandle <> NIL THEN
        BEGIN
            GetResInfo(myHandle, myID, myType, myName); {get resource name}
            DetachResource(myHandle); {detach resource}
            UseResFile(dest);         {set destination resource file}
            AddResource(myHandle, theType, theID, myName);
            IF ResError = noErr THEN
                WriteResource(myHandle); {write resource data}
            END;
            DoCopyResource := ResError; {return result code}
            ReleaseResource(myHandle);
        END;
    END;
END;

```

See the chapter “Resource Manager” in *Inside Macintosh: More Macintosh Toolbox* for details about the routines used in Listing 3-13.

Reverting to a Saved File

Many applications that manipulate files provide a menu command that allows the user to revert to the last saved version of a document. The technique for handling this command is relatively simple. First you should display a dialog box asking whether to revert to the last saved version of the file, as illustrated in Figure 3-11.

Figure 3-11 A Revert to Saved dialog box

If the user clicks the Cancel button, nothing should happen to the current document. If, however, the user confirms the menu command by clicking OK, you just need to call `DoReadFile` to read the disk version of the file back into the window. Listing 3-14 illustrates how to implement a Revert to Saved menu command.

Listing 3-14 Handling the Revert to Saved menu command

```

FUNCTION DoRevertCmd: OSErr;
VAR
    myWindow:    WindowPtr;           {window for file data}
    myData:      MyDocRecHnd;         {handle to window data}
    myFile:      Integer;             {file reference number}
    myName:      Str255;              {file's name}
    myDialog:    DialogPtr;           {pointer to modal dialog box}
    myItem:      Integer;             {item selected in modal dialog}
    myPort:      GrafPtr;            {the original graphics port}
CONST
    kRevertDialog = 128;              {resource ID of Revert to Saved dialog}
BEGIN
    myWindow := FrontWindow;          {get pointer to front window}
                                     {get handle to window's data record}
    myData := MyDocRecHnd(GetWRefCon(myWindow));
    GetWTitle(myWindow, myName);      {get file's name}
    ParamText(myName, '', '', '');
    myDialog := GetNewDialog(kRevertDialog, NIL, WindowPtr(-1));
    GetPort(myPort);
    SetPort(myDialog);

    REPEAT
        ModalDialog(NIL, myItem);
    UNTIL (myItem = iOK) OR (myItem = iCancel);

    DisposeDialog(myDialog);
    SetPort(myPort);                  {restore previous grafPort}

    IF myItem = iOK THEN
        DoRevertCmd := DoReadFile(myWindow);
    ELSE
        DoRevertCmd := noErr;
    END;
END;

```

The `DoRevertCmd` function retrieves the document record handle from the frontmost window's reference constant field and then gets the window's title (which is also the name of the file) and inserts it into a modal dialog box.

If the user clicks the OK button, `DoRevertCmd` calls the `DoReadFile` function to read the data from the file into the window. Otherwise, `DoRevertCmd` simply exits without changing the data in the window.

Closing a File

In most cases, your application closes a file after a user clicks in a window's close box or chooses the Close command in the File menu. The Close menu command should be active only when there is actually an active window on the desktop. If there is an active window, you need to determine whether it belongs to your application; if so, you need to handle dialog windows and document windows differently, as illustrated in Listing 3-15.

Listing 3-15 Handling the Close menu command

```
FUNCTION DoCloseCmd: OSErr;
VAR
    myWindow:    WindowPtr;
    myData:      MyDocRecHnd;
    myErr:       OSErr;
BEGIN
    myErr := FALSE;
    myWindow := FrontWindow;           {get window to be closed}
    CASE MyGetWindowType(myWindow) OF
        kDAWindow:
            CloseDeskAcc(WindowPeek(myWindow)^.windowKind);
        kMyModelessDialog:
            HideWindow(myWindow);       {for dialogs, hide the window}
        kMyDocWindow:
            BEGIN
                myData := MyDocRecHnd(GetWRefCon(myWindow));
                myErr := DoCloseFile(myData);
                IF myErr = noErr THEN
                    DisposeWindow(myWindow);
            END;
        OTHERWISE
            ;
    END;
    DoCloseCmd := myErr;
END;
```

The `DoCloseCmd` function determines the type of the frontmost window by calling the application-defined function `MyGetWindowType`. (See the chapter “Window Manager” in *Inside Macintosh: Macintosh Toolbox Essentials* for a definition of `MyGetWindowType`.) If the window to be closed is a window belonging to a desk accessory, `DoCloseCmd` closes

the desk accessory. If the window to be closed is a dialog window, this procedure just hides the window. If the window to be closed is a document window, `DoCloseCmd` retrieves its document record handle and calls both `DoCloseFile` (defined in Listing 3-16) and `DisposeWindow`. Before you close the file associated with a window, you should check whether the contents of the window have changed since the last time the document was saved. If so, you should ask the user whether to save those changes. Listing 3-16 illustrates one way to do this.

Listing 3-16 Closing a file

```

FUNCTION DoCloseFile (myData: MyDocRecHnd): OSErr;
VAR
    myErr:      OSErr;
    myDialog:   DialogPtr;      {pointer to modal dialog box}
    myItem:     Integer;        {item selected in alert box}
    myPort:     GrafPtr;        {the original graphics port}
CONST
    kSaveChangesDialog = 129;   {resource of Save changes dialog}
BEGIN
    IF myData^.windowDirty THEN {see whether window is dirty}
    BEGIN
        myItem := CautionAlert(kSaveChangesDialog, NIL);
        IF myItem = iCancel THEN {user clicked Cancel}
        BEGIN
            DoCloseFile := usrCanceledErr;
            Exit(DoCloseFile);
        END;
        IF myItem = iSave THEN
            myErr := DoSaveCmd;
        END;
    IF myData^.fileRefNum <> 0 THEN
    BEGIN
        myErr := FSClose(myData^.fileRefNum);
        IF myErr = noErr THEN
        BEGIN
            myErr := FlushVol(NIL, myData^.fileFSSpec.vRefNum);
            myData^.fileRefNum := 0; {clear the file reference number}
        END;
    END;
    {Dispose of TextEdit record and controls here (code omitted).}
    DisposeHandle(Handle(myData)); {dispose of document record}
    DoCloseFile := myErr;
END;

```

If the document is an existing file that has not been changed since it was last saved, your application can simply call the `FSClose` function. This routine writes to disk any unwritten data remaining in the volume buffer. The `FSClose` function also updates the information maintained on the volume for that file and removes the access path. The information about the file is not actually written to the disk, however, until the volume is flushed, ejected, or unmounted. To keep the file information current, it's a good idea to follow each call to `FSClose` with a call to the `FlushVol` function.

If the contents of an existing file have been changed, or if a new file is being closed for the first time, your application can call the Dialog Manager routine `CautionAlert` (specifying a resource ID of an 'ALRT' template) to ask the user whether or not to save the changes. If the user decides not to save the file, you can just call `FSClose` and dispose of the window. Otherwise, `DoCloseFile` calls the `DoSaveCmd` function to save the file to disk.

Opening Files at Application Startup Time

A user often launches your application by double-clicking one of its document icons or by selecting one or more document icons and choosing the Open command in the Finder's File menu. In these cases, your application needs to determine which files the user selected so that it can open each one and display its contents in a window. There are two ways in which your application can determine this.

If the user opens a file from the Finder and if your application supports high-level events, the Finder sends it an Open Documents event. Your application then needs to determine which file or files to open and react accordingly. For a complete description of how to process the Open Documents event, see the chapter "Apple Event Manager" in *Inside Macintosh: Interapplication Communication*.

IMPORTANT

If at all possible, your application should support high-level events. You should use the techniques illustrated in this section only if your application doesn't support high-level events. ▲

If your application does not support high-level events, you need to ask the Finder at application launch time whether or not the user launched the application by selecting some documents. You can do this by calling the `CountAppFiles` procedure and seeing whether the count of files is 1 or more. Then you can call the procedures `GetAppFiles` and `ClrAppFiles` to retrieve the information about the selected files. The technique is illustrated in Listing 3-17.

The `CountAppFiles` procedure determines how many files, if any, the user selected at application startup time. If the value of the `myNum` parameter is nonzero, then `myJob` contains a value that indicates whether the files were selected for opening or printing. Currently, `myJob` can have one of two values:

```
CONST
    appOpen  = 0;    {open the document(s)}
    appPrint = 1;    {print the document(s)}
```

Listing 3-17 Opening files at application launch time

```

PROCEDURE DoInitFiles;
VAR
    myNum: Integer;    {number of files to be opened or printed}
    myJob: Integer;    {open or print the files?}
    index: Integer;    {index of current file}
    myFile: AppFile;   {file info}
    mySpec: FSSpec;    {file system specification}
    myErr: OSErr;
BEGIN
    CountAppFiles(myJob, myNum);
    IF myNum > 0 THEN                                     {user selected some files}
        IF myJob = appOpen THEN                           {files are to be opened}
            FOR index := 1 TO myNum DO
                BEGIN
                    GetAppFiles(index, myFile);    {get file info from Finder}
                    myErr := FSMakeFSSpec(myFile.vRefNum, 0, myFile.fName,
                                           mySpec); {make an FSSpec to hold info}
                    myErr := DoOpenFile(mySpec);   {read in file's data}
                    ClrAppFiles(index);            {show we've got the info}
                END;
            END;
        END;
    END;

```

In Listing 3-17, if the files are to be opened, then `DoInitFiles` obtains information about them by calling the `GetAppFiles` procedure for each one. The `GetAppFiles` procedure returns the information in a record of type `AppFile`.

```

TYPE AppFile =
    RECORD
        vRefNum: Integer;    {working directory reference number}
        fType: OSType;      {file type}
        versNum: Integer;    {version number; ignored}
        fName: Str255;       {filename}
    END;

```

Because the function `DoOpenFile` takes an `FSSpec` record as a parameter, `DoInitFiles` next converts the information returned in the `myFile` parameter into an `FSSpec` record, using `FSMakeFSSpec`. Then `DoInitFiles` calls `DoOpenFile` to read the file data and `ClrAppFiles` to let the Finder know that it has processed the information for that file.

Note

The `vRefNum` field of an `AppFile` record does not contain a volume reference number; instead it contains a working directory reference number, which encodes both the volume reference number and the parent directory ID. (That's why the second parameter passed to `FSMakeFSSpec` in Listing 3-17 is 0.) ♦

Using a Preferences File

Many applications allow the user to alter various settings that control the operation or configuration of the application. For example, your application might allow the user to specify the size and placement of any new windows or the default font used to display text in those windows. You can create a preferences file in which to record user preferences, and your application can retrieve that file whenever it is launched.

In deciding how to structure your preferences file, it is important to distinguish document-specific settings from application-specific settings. Some user-specifiable settings affect only a particular document. For example, the user might have changed the text font in a particular window. When you save the text in the window, you also want to save the current font setting. Generally you can do this by storing the font name in a resource in the document file's resource fork. Then, when the user opens that document again, you check for the presence of such a resource, retrieve the information stored in it, and set the document font accordingly.

Some settings, such as a default text font, are not specific to a particular document. You might store such settings in the application's resource fork, but generally it is better to store them in a separate preferences file. The main reason for this is to avoid problems that can arise if an application is located on a server volume. If preferences are stored in resources in the application's resource fork, those preferences apply to all users executing that application. Worse yet, the resources can become corrupted if several different users attempt to alter the settings at the same time.

Thus, it is best to store application-specific settings in a preferences file. The Operating System provides a special folder in the System Folder, called Preferences, where you can store that file. Listing 3-18 illustrates a way to open your application's preferences file.

Listing 3-18 Opening a preferences file

```
PROCEDURE DoGetPreferences;
VAR
    myErr:      OSErr;
    myVRef:     Integer; {volume ref num of Preferences folder}
    myDirID:    LongInt; {dir ID of Preferences folder}
    mySpec:     FSSpec;  {FSSpec for the preferences file}
    myName:     Str255;  {name of the application}
    myRef:      Integer; {ref num of app's resource file; ignored}
    myHand:     Handle;  {handle to Finder information; ignored}
    myRefNum:   Integer; {file reference number}
```

```

CONST
    kPrefID = 128;          {resource ID of STR# with filename}
BEGIN
    {Determine the name of the preferences file.}
    GetIndString(myName, kPrefID, 1);

    {Find the Preferences folder in the System Folder.}
    myErr := FindFolder(kOnSystemDisk, kPreferencesFolderType,
                        kDontCreateFolder, myVRef, myDirID);
    IF myErr = noErr THEN
        myErr := FSMakeFSSpec(myVRef, myDirID, myName, mySpec);
    IF myErr = noErr THEN
        myRefNum := FSpOpenResFile(mySpec, fsCurPerm);

    {Read your preference settings here.}

    CloseResFile(myRefNum);
END;

```

The `DoGetPreferences` procedure first determines the name of the preferences file it is to open and read. To allow easy localization, you should store the name in a resource of type 'STR#' in your application's resource file. The `DoGetPreferences` procedure assumes that the name is stored as the first string in the resource having ID `kPrefID`.

The technique shown here assumes that your preference settings can all be stored in resources. As a result, Listing 3-18 calls the Resource Manager function `FSpOpenResFile` to open the resource fork of your preferences file. See the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for complete details on opening resource files and reading resources from them.

Adjusting the File Menu

Your application should dim any File menu commands that are not available at the time the user pulls down the File menu. For example, if your application does not yet have a document window open, then the Save, Save As, and Revert commands should be dimmed. You can adjust the File menu easily using the technique shown in Listing 3-19.

Listing 3-19 Adjusting the File menu

```

PROCEDURE DoAdjustFileMenu;
VAR
    myWindow:   WindowPtr;
    myMenu:     MenuHandle;
    myData:     MyDocRecHnd;           {handle to window data}

```

Introduction to File Management

```

BEGIN
    myWindow := FrontWindow;
    IF myWindow = NIL THEN
        BEGIN
            myMenu := GetMHandle(mFile);
            DisableItem(myMenu, iSave);           {disable Save}
            DisableItem(myMenu, iSaveAs);         {disable Save As}
            DisableItem(myMenu, iRevert);         {disable Revert}
            DisableItem(myMenu, iClose);          {disable Close}
        END
    ELSE IF MyGetWindowType(myWindow) = kMyDocWindow THEN
        BEGIN
            myData := MyDocRecHnd(GetWRefCon(myWindow));
            myMenu := GetMHandle(mFile);
            EnableItem(myMenu, iSaveAs);          {enable Save As}
            EnableItem(myMenu, iClose);           {enable Close}

            IF myData^.windowDirty THEN
                BEGIN
                    EnableItem(myMenu, iSave);     {enable Save}
                    EnableItem(myMenu, iRevert);   {enable Revert}
                END
            ELSE
                BEGIN
                    DisableItem(myMenu, iSave);    {disable Save}
                    DisableItem(myMenu, iRevert);  {disable Revert}
                END;
            END;
        END;
    END;
END;

```

Your application should call `DoAdjustFileMenu` whenever it receives a mouse-down event in the menu bar. (No doubt you want to include code appropriate for enabling and disabling other menu items too.) See the chapter “Menu Manager” in *Inside Macintosh: Macintosh Toolbox Essentials* for details on the menu enabling and disabling procedures used in Listing 3-19.

File Management Reference

This section describes the data structures and routines used in this chapter to illustrate basic file management operations. The section “Data Structures” shows the Pascal data structures for the file system specification record and the standard file reply record. The sections that follow describe the Standard File Package routines for opening and saving

documents and the File Manager routines for accessing files, manipulating files and directories, accessing volumes, and getting information about documents to be opened when your application is launched.

For a description of other file-related data structures and routines, see the chapters “File Manager” and “Standard File Package” in this book.

Data Structures

This section describes the data structures that your application can use to exchange information with the File Manager and the Standard File Package. The techniques described in this chapter use file system specification records and standard file reply records.

File System Specification Record

The file system specification record for files and directories is defined by the `FSSpec` data type.

```
TYPE FSSpec =                                {file system specification}
RECORD
    vRefNum:   Integer;    {volume reference number}
    parID:     LongInt;    {directory ID of parent directory}
    name:      Str63;      {filename or directory name}
END;
```

Field descriptions

<code>vRefNum</code>	The volume reference number of the volume containing the specified file or directory.
<code>parID</code>	The directory ID of the directory containing the specified file or directory.
<code>name</code>	The name of the specified file or directory.

Standard File Reply Records

The procedures `StandardGetFile` and `StandardPutFile` both return information to your application using a standard file reply record, which is defined by the `StandardFileReply` data type. The reply record identifies selected files with a file system specification record, which you can pass directly to many of the File Manager functions described in the sections that follow. The reply record also contains fields that support several Finder features.

Introduction to File Management

```

TYPE StandardFileReply =
RECORD
    sfGood:          Boolean;      {TRUE if user did not cancel}
    sfReplacing:     Boolean;      {TRUE if replacing file with same name}
    sfType:          OSType;       {file type}
    sfFile:          FSSpec;       {selected file, folder, or volume}
    sfScript:        ScriptCode;   {script of file, folder, or volume name}
    sfFlags:         Integer;      {Finder flags of selected item}
    sfIsFolder:      Boolean;      {selected item is a folder}
    sfIsVolume:      Boolean;      {selected item is a volume}
    sfReserved1:     LongInt;      {reserved}
    sfReserved2:     Integer;      {reserved}
END;

```

Field descriptions

<code>sfGood</code>	Reports whether the reply record is valid. The value is <code>TRUE</code> after the user clicks Save or Open; <code>FALSE</code> after the user clicks Cancel. When the user has completed the dialog box, the other fields in the reply record are valid only if the <code>sfGood</code> field contains <code>TRUE</code> .
<code>sfReplacing</code>	Reports whether a file to be saved replaces an existing file of the same name. This field is valid only after a call to the <code>StandardPutFile</code> or <code>CustomPutFile</code> procedure. When the user assigns a name that duplicates that of an existing file, the Standard File Package asks for verification by displaying a subsidiary dialog box (illustrated in Figure 3-10). If the user verifies the name, the Standard File Package sets the <code>sfReplacing</code> field to <code>TRUE</code> and returns to your application; if the user cancels the overwriting of the file, the Standard File Package returns to the main dialog box. If the name does not conflict with an existing name, the Standard File Package sets the field to <code>FALSE</code> and returns.
<code>sfType</code>	Contains the file type of the selected file. (File types are described in the chapter “Finder Interface” in <i>Inside Macintosh: Macintosh Toolbox Essentials</i> .) Only <code>StandardGetFile</code> and <code>CustomGetFile</code> return a file type in this field.
<code>sfFile</code>	Describes the selected file, folder, or volume with a file system specification record, which contains a volume reference number, parent directory ID, and name. (See the chapter “File Manager” in this book for a complete description of the file system specification record.) If the selected item is an alias for another item, the Standard File Package resolves the alias and places the file system specification record for the target in the <code>sfFile</code> field when the user completes the dialog box. If the selected file is a stationery pad, the reply record describes the file itself, not a copy of the file.
<code>sfScript</code>	Identifies the script in which the name of the document is to be displayed. (This information is used by the Finder and by the Standard File Package.) A script code of <code>smSystemScript</code> (-1) represents the default system script.

<code>sfFlags</code>	Contains the Finder flags from the Finder information record in the catalog entry for the selected file. (See the chapter “Finder Interface” in <i>Inside Macintosh: Macintosh Toolbox Essentials</i> for a description of the Finder flags.) This field is returned only by <code>StandardGetFile</code> and <code>CustomGetFile</code> . If your application supports stationery, it should check the stationery bit in the Finder flags to determine whether to treat the selected file as stationery. Unlike the Finder, the Standard File Package does not automatically create a document from a stationery pad and pass your application the new document. If the user opens a stationery document from within an application that does not support stationery, the Standard File Package displays a dialog box warning the user that the master copy is being opened.
<code>sfIsFolder</code>	Reports whether the selected item is a folder (TRUE) or a file or volume (FALSE). This field is meaningful only during the execution of a dialog hook function.
<code>sfIsVolume</code>	Reports whether the selected item is a volume (TRUE) or a file or folder (FALSE). This field is meaningful only during the execution of a dialog hook function.
<code>sfReserved1</code>	Reserved.
<code>sfReserved2</code>	Reserved.

Application Files Records

The `GetAppFiles` procedure returns information about files opened at application launch time in an application files record, defined by the `AppFile` data type:

```

TYPE AppFile =
RECORD
    vRefNum:   Integer;    {working directory reference number}
    fType:     OSType;     {file type}
    versNum:   Integer;    {version number; ignored}
    fName:     Str255;     {filename}
END;
```

Field descriptions

<code>vRefNum</code>	A working directory reference number that encodes the volume and parent directory of the file.
<code>fType</code>	The file type.
<code>versNum</code>	Reserved.
<code>fName</code>	The filename.

File Specification Routines

If your application has no special user interface requirements, you can use the `StandardGetFile` and `StandardPutFile` procedures to display the default dialog boxes for opening and saving documents. For a description of more advanced file specification routines, see the chapter “Standard File Package” in this book.

StandardGetFile

You can use the `StandardGetFile` procedure to display the default Open dialog box when the user is opening a file.

```
PROCEDURE StandardGetFile (fileFilter: FileFilterProcPtr;
                           numTypes: Integer;
                           typeList: SFTypeList;
                           VAR reply: StandardFileReply);
```

<code>fileFilter</code>	A pointer to an optional file filter function, provided by your application, through which <code>StandardGetFile</code> passes files of the specified types.
<code>numTypes</code>	The number of file types to be displayed. If you specify a <code>numTypes</code> value of -1, the first filtering passes files of all types.
<code>typeList</code>	A list of file types to be displayed.
<code>reply</code>	The reply record, which <code>StandardGetFile</code> fills in before returning.

DESCRIPTION

The `StandardGetFile` procedure presents a dialog box through which the user specifies the name and location of a file to be opened. While the dialog box is active, `StandardGetFile` gets and handles events until the user completes the interaction, either by selecting a file to open or by canceling the operation. `StandardGetFile` returns the user's input in a record of type `StandardFileReply`.

The `fileFilter`, `numTypes`, and `typeList` parameters together determine which files appear in the displayed list. The first filtering is by file type, which you specify in the `numTypes` and `typeList` parameters. The `numTypes` parameter specifies the number of file types to be displayed. You can specify one or more types. If you specify a `numTypes` value of -1, the first filtering passes files of all types.

The `fileFilter` parameter points to an optional file filter function, provided by your application, through which `StandardGetFile` passes files of the specified types. See the chapter “Standard File Package” in this book for a complete description of how you specify this filter function.

SPECIAL CONSIDERATIONS

The `StandardGetFile` procedure is not available in all versions of system software. Use the `Gestalt` function to determine whether `StandardGetFile` is available before calling it.

Because `StandardGetFile` may move memory, you should not call it at interrupt time.

StandardPutFile

You can use the `StandardPutFile` procedure to display the default Save dialog box when the user is saving a file.

```
PROCEDURE StandardPutFile (prompt: Str255; defaultName: Str255;
                           VAR reply: StandardFileReply);
```

`prompt` The prompt message to be displayed over the text field.

`defaultName` The initial name of the file.

`reply` The reply record, which `StandardPutFile` fills in before returning.

DESCRIPTION

The `StandardPutFile` procedure presents a dialog box through which the user specifies the name and location of a file to be written to. The dialog box is centered on the screen. While the dialog box is active, `StandardPutFile` gets and handles events until the user completes the interaction, either by selecting a name and authorizing the save or by canceling the save. The `StandardPutFile` procedure returns the user's input in a record of type `StandardFileReply`.

SPECIAL CONSIDERATIONS

The `StandardPutFile` procedure is not available in all versions of system software. Use the `Gestalt` function to determine whether `StandardPutFile` is available before calling it.

Because `StandardPutFile` may move memory, you should not call it at interrupt time.

File Access Routines

This section describes the File Manager's file access routines. When you call one of these routines, you specify a file by a path reference number (which the File Manager returns to your application when your application opens the file). Unless your application has very specialized needs, you should be able to manage all file access (for example, writing data to the file) using the routines described in this section. Typically you use these routines to operate on a file's data fork, but in certain circumstances you might want to use them on a file's resource fork as well.

Reading, Writing, and Closing Files

You can use the functions `FSRead`, `FSWrite`, and `FSClose` to read data from a file, write data to a file, and close an open file. All three of these functions operate on open files. You can use any one of a variety of routines to open a file (for example, `FSpOpenDF`).

FSRead

You can use the `FSRead` function to read any number of bytes from an open file.

```
FUNCTION FSRead (refNum: Integer; VAR count: LongInt;
                buffPtr: Ptr): OSErr;
```

<code>refNum</code>	The file reference number of an open file.
<code>count</code>	On input, the number of bytes to read; on output, the number of bytes actually read.
<code>buffPtr</code>	A pointer to the data buffer into which the bytes are to be read.

DESCRIPTION

The `FSRead` function attempts to read the requested number of bytes from the specified file into the specified buffer. The `buffPtr` parameter points to that buffer; this buffer is allocated by your application and must be at least as large as the `count` parameter.

Because the read operation begins at the current mark, you might want to set the mark first by calling the `SetFPos` function. If you try to read past the logical end-of-file, `FSRead` reads in all the data up to the end-of-file, moves the mark to the end-of-file, and returns `eofErr` as its function result. Otherwise, `FSRead` moves the file mark to the byte following the last byte read and returns `noErr`.

RESULT CODES

<code>noErr</code>	0	No error
<code>ioErr</code>	-36	I/O error
<code>fnOpnErr</code>	-38	File not open
<code>eofErr</code>	-39	Logical end-of-file reached
<code>posErr</code>	-40	Attempt to position mark before start of file
<code>fLckdErr</code>	-45	File is locked
<code>paramErr</code>	-50	Negative count
<code>rfNumErr</code>	-51	Bad reference number
<code>afpAccessDenied</code>	-5000	User does not have the correct access to the file

FSWrite

You can use the `FSWrite` function to write any number of bytes to an open file.

```
FUNCTION FSWrite (refNum: Integer; VAR count: LongInt;
                 buffPtr: Ptr): OSErr;
```

<code>refNum</code>	The file reference number of an open file.
<code>count</code>	On input, the number of bytes to write to the file; on output, the number of bytes actually written.
<code>buffPtr</code>	A pointer to the data buffer from which the bytes are to be written.

DESCRIPTION

The `FSWrite` function takes the specified number of bytes from the specified data buffer and attempts to write them to the specified file. Because the write operation begins at the current mark, you might want to set the mark first by calling the `SetFPos` function.

If the write operation completes successfully, `FSWrite` moves the file mark to the byte following the last byte written and returns `noErr`. If you try to write past the logical end-of-file, `FSWrite` moves the logical end-of-file. If you try to write past the physical end-of-file, `FSWrite` adds one or more clumps to the file and moves the physical end-of-file accordingly.

RESULT CODES

<code>noErr</code>	0	No error
<code>dskFulErr</code>	-34	Disk full
<code>ioErr</code>	-36	I/O error
<code>fnOpnErr</code>	-38	File not open
<code>posErr</code>	-40	Attempt to position mark before start of file
<code>wPrErr</code>	-44	Hardware volume lock
<code>fLckdErr</code>	-45	File is locked
<code>vLckdErr</code>	-46	Software volume lock
<code>paramErr</code>	-50	Negative count
<code>rfNumErr</code>	-51	Bad reference number
<code>wrPermErr</code>	-61	Read/write permission doesn't allow writing

FSClose

You can use the `FSClose` function to close an open file.

```
FUNCTION FSClose (refNum: Integer): OSErr;
```

<code>refNum</code>	The file reference number of an open file.
---------------------	--

DESCRIPTION

The `FSClose` function removes the access path for the specified file and writes the contents of the volume buffer to the volume.

Note

The `FSClose` function calls `PBFlushFile` internally to write the file's bytes onto the volume. To ensure that the file's catalog entry is updated, you should call `FlushVol` after you call `FSClose`. ♦

▲ WARNING

Make sure that you do not call `FSClose` with a file reference number of a file that has already been closed. Attempting to close the same file twice may result in loss of data on a volume. See the description of file control blocks in the chapter "File Manager" in this book for a discussion of how this can happen. ▲

RESULT CODES

<code>noErr</code>	0	No error
<code>ioErr</code>	-36	I/O error
<code>fnOpnErr</code>	-38	File not open
<code>fnfErr</code>	-43	File not found
<code>rfNumErr</code>	-51	Bad reference number

Manipulating the File Mark

You can use the functions `GetFPos` and `SetFPos` to get or set the current position of the file mark.

GetFPos

You can use the `GetFPos` function to determine the current position of the mark before reading from or writing to an open file.

```
FUNCTION GetFPos (refNum: Integer; VAR filePos: LongInt): OSErr;
```

`refNum` The file reference number of an open file.
`filePos` On output, the current position of the mark.

DESCRIPTION

The `GetFPos` function returns, in the `filePos` parameter, the current position of the file mark for the specified open file. The position value is zero-based; that is, the value of `filePos` is 0 if the file mark is positioned at the beginning of the file.

RESULT CODES

noErr	0	No error
ioErr	-36	I/O error
fnOpnErr	-38	File not open
rfNumErr	-51	Bad reference number
gfpErr	-52	Error during GetFPos

SetFPos

You can use the `SetFPos` function to set the position of the file mark before reading from or writing to an open file.

```
FUNCTION SetFPos (refNum: Integer; posMode: Integer;
                 posOff: LongInt): OSErr;
```

refNum	The file reference number of an open file.
posMode	The positioning mode.
posOff	The positioning offset.

DESCRIPTION

The `SetFPos` function sets the file mark of the specified file. The `posMode` parameter indicates how to position the mark; it must contain one of the following values:

```
CONST
    fsAtMark      = 0; {at current mark}
    fsFromStart   = 1; {set mark relative to beginning of file}
    fsFromLEOF    = 2; {set mark relative to logical end-of-file}
    fsFromMark    = 3; {set mark relative to current mark}
```

If you specify `fsAtMark`, the mark is left wherever it's currently positioned, and the `posOff` parameter is ignored. The next three constants let you position the mark relative to either the beginning of the file, the logical end-of-file, or the current mark. If you specify one of these three constants, you must also pass in `posOff` a byte offset (either positive or negative) from the specified point. If you specify `fsFromLEOF`, the value in `posOff` must be less than or equal to 0.

RESULT CODES

noErr	0	No error
ioErr	-36	I/O error
fnOpnErr	-38	File not open
eofErr	-39	Logical end-of-file reached
posErr	-40	Attempt to position mark before start of file
rfNumErr	-51	Bad reference number

Manipulating the End-of-File

You can use the functions `GetEOF` and `SetEOF` to get or set the logical end-of-file of an open file.

GetEOF

You can use the `GetEOF` function to determine the current logical end-of-file of an open file.

```
FUNCTION GetEOF (refNum: Integer; VAR logEOF: LongInt): OSerr;
```

`refNum` The file reference number of an open file.

`logEOF` On output, the logical end-of-file.

DESCRIPTION

The `GetEOF` function returns, in the `logEOF` parameter, the logical end-of-file of the specified file.

RESULT CODES

<code>noErr</code>	0	No error
<code>ioErr</code>	-36	I/O error
<code>fnOpnErr</code>	-38	File not open
<code>rfNumErr</code>	-51	Bad reference number
<code>afpAccessDenied</code>	-5000	User does not have the correct access to the file

SEE ALSO

For a description of the logical and physical end-of-file, see the section “File Access Characteristics” on page 3-122.

SetEOF

You can use the `SetEOF` function to set the logical end-of-file of an open file.

```
FUNCTION SetEOF (refNum: Integer; logEOF: LongInt): OSerr;
```

`refNum` The file reference number of an open file.

`logEOF` The logical end-of-file.

DESCRIPTION

The `SetEOF` function sets the logical end-of-file of the specified file. If you attempt to set the logical end-of-file beyond the physical end-of-file, the physical end-of-file is set 1 byte beyond the end of the next free allocation block; if there isn't enough space on the volume, no change is made, and `SetEOF` returns `dskFulErr` as its function result.

If you set the `logEOF` parameter to 0, all space occupied by the file on the volume is released. The file still exists, but it contains 0 bytes. Setting a file fork's end-of-file to 0 is therefore not the same as deleting the file (which removes both file forks at once).

RESULT CODES

<code>noErr</code>	0	No error
<code>dskFulErr</code>	-34	Disk full
<code>ioErr</code>	-36	I/O error
<code>fnOpnErr</code>	-38	File not open
<code>wPrErr</code>	-44	Hardware volume lock
<code>fLckdErr</code>	-45	File is locked
<code>vLckdErr</code>	-46	Software volume lock
<code>rfNumErr</code>	-51	Bad reference number
<code>wrPermErr</code>	-61	Read/write permission doesn't allow writing

SEE ALSO

For a description of the logical and physical end-of-file, see the section “File Access Characteristics” on page 3-122.

File and Directory Manipulation Routines

The File Manager includes a set of file and directory manipulation routines that accept `FSSpec` records as parameters. Depending on the requirements of your application and on the environment in which it is running, you may be able to accomplish all your file and directory operations by using these routines.

Before calling any of these routines, however, you should call the `Gestalt` function to ensure that they are available in the operating environment. (See “Testing for File Management Routines” on page 3-128 for an illustration of calling `Gestalt`.) If these routines are not available, you can call the corresponding HFS routines.

Opening, Creating, and Deleting Files

The File Manager provides the `FSpOpenDF`, `FSpCreate`, and `FSpDelete` routines, which allow you to open, create, and delete files.

FSpOpenDF

You can use the FSpOpenDF function to open a file's data fork.

```
FUNCTION FSpOpenDF (spec: FSSpec; permission: SignedByte;
                   VAR refNum: Integer): OSErr;
```

spec An FSSpec record specifying the file whose data fork is to be opened.

permission A constant indicating the desired file access permissions.

refNum A reference number of an access path to the file's data fork.

DESCRIPTION

The FSpOpenDF function opens the data fork of the file specified by the `spec` parameter and returns a file reference number in the `refNum` parameter. You can pass that reference number as a parameter to any of the low- or high-level file access routines.

The `permission` parameter specifies the kind of access permission mode you want. You can specify one of these constants:

```
CONST
    fsCurPerm      = 0;    {whatever permission is allowed}
    fsRdPerm        = 1;    {read permission}
    fsWrPerm        = 2;    {write permission}
    fsRdWrPerm      = 3;    {exclusive read/write permission}
    fsRdWrShPerm    = 4;    {shared read/write permission}
```

In most cases, you can simply set the `permission` parameter to `fsCurPerm`. Some applications request `fsRdWrPerm`, to ensure that they can both read from and write to a file.

RESULT CODES

<code>noErr</code>	0	No error
<code>nsvErr</code>	-35	No such volume
<code>ioErr</code>	-36	I/O error
<code>bdNamErr</code>	-37	Bad filename
<code>tmfoErr</code>	-42	Too many files open
<code>fnfErr</code>	-43	File not found
<code>opWrErr</code>	-49	File already open for writing
<code>permErr</code>	-54	Attempt to open locked file for writing
<code>dirNFErr</code>	-120	Directory not found or incomplete pathname
<code>afpAccessDenied</code>	-5000	User does not have the correct access to the file

FSpCreate

You can use the `FSpCreate` function to create a new file.

```
FUNCTION FSpCreate (spec: FSSpec; creator: OSType;
                  fileType: OSType; scriptTag: ScriptCode):
    OSErr;
```

<code>spec</code>	An <code>FSSpec</code> record specifying the file to be created.
<code>creator</code>	The creator of the new file.
<code>fileType</code>	The file type of the new file.
<code>scriptTag</code>	The code of the script system in which the filename is to be displayed. If you have established the name and location of the new file using either the <code>StandardPutFile</code> or <code>CustomPutFile</code> procedure, specify the script code returned in the reply record. (See the chapter “Standard File Package” in this book for a description of <code>StandardPutFile</code> and <code>CustomPutFile</code> .) Otherwise, specify the system script by setting the <code>scriptTag</code> parameter to the value <code>smSystemScript</code> .

DESCRIPTION

The `FSpCreate` function creates a new file (both forks) with the specified type, creator, and script code. The new file is unlocked and empty. The date and time of creation and last modification are set to the current date and time.

See the chapter “Finder Interface” in *Inside Macintosh: Macintosh Toolbox Essentials* for information on file types and creators.

Files created using `FSpCreate` are not automatically opened. If you want to write data to the new file, you must first open the file using a file access routine (such as `FSpOpenDF`).

Note

The resource fork of the new file exists but is empty. You’ll need to call one of the Resource Manager procedures `CreateResFile`, `HCreateResFile`, or `FSpCreateResFile` to create a resource map in the file before you can open it (by calling one of the Resource Manager functions `OpenResFile`, `HOpenResFile`, or `FSpOpenResFile`). ♦

RESULT CODES

noErr	0	No error
dirFulErr	-33	File directory full
dskFulErr	-34	Disk is full
nsvErr	-35	No such volume
ioErr	-36	I/O error
bdNamErr	-37	Bad filename
fnfErr	-43	Directory not found or incomplete pathname
wPrErr	-44	Hardware volume lock
vLckdErr	-46	Software volume lock
dupFNerr	-48	Duplicate filename and version
dirNFerr	-120	Directory not found or incomplete pathname
afpAccessDenied	-5000	User does not have the correct access
afpObjectTypeErr	-5025	A directory exists with that name

FSpDelete

You can use the `FSpDelete` function to delete files and directories.

```
FUNCTION FSpDelete (spec: FSSpec): OSErr;
```

`spec` An `FSSpec` record specifying the file or directory to delete.

DESCRIPTION

The `FSpDelete` function removes a file or directory. If the specified target is a file, both forks of the file are deleted. The file ID reference, if any, is removed.

A file must be closed before you can delete it. Similarly, a directory must be empty before you can delete it. If you attempt to delete an open file or a nonempty directory, `FSpDelete` returns the result code `fBsyErr`. `FSpDelete` also returns the result code `fBsyErr` if the directory has an open working directory associated with it.

RESULT CODES

noErr	0	No error
nsvErr	-35	No such volume
ioErr	-36	I/O error
bdNamErr	-37	Bad filename
fnfErr	-43	File not found
wPrErr	-44	Hardware volume lock
fLckdErr	-45	File is locked
vLckdErr	-46	Software volume lock
fBsyErr	-47	File busy, directory not empty, or working directory control block open
dirNFerr	-120	Directory not found or incomplete pathname
afpAccessDenied	-5000	User does not have the correct access

Exchanging the Data in Two Files

The function `FSpExchangeFiles` allows you to exchange the data in two files.

FSpExchangeFiles

You can use the `FSpExchangeFiles` function to exchange the data stored in two files on the same volume.

```
FUNCTION FSpExchangeFiles (source: FSSpec; dest: FSSpec): OSErr;
```

source The source file. The contents of this file and its file information are placed in the file specified by the **dest** parameter.

dest The destination file. The contents of this file and its file information are placed in the file specified by the **source** parameter.

DESCRIPTION

The `FSpExchangeFiles` function swaps the data in two files by changing the information in the volume's catalog and, if the files are open, in the file control blocks.

You should use `FSpExchangeFiles` when updating an existing file, so that the file ID remains valid in case the file is being tracked through its file ID. The `FSpExchangeFiles` function changes the fields in the catalog entries that record the location of the data and the modification dates. It swaps both the data forks and the resource forks.

The `FSpExchangeFiles` function works on both open and closed files. If either file is open, `FSpExchangeFiles` updates any file control blocks associated with the file.

Exchanging the contents of two files requires essentially the same access permissions as opening both files for writing.

The files whose data is to be exchanged must both reside on the same volume. If they do not, `FSpExchangeFiles` returns the result code `diffVolErr`.

RESULT CODES

<code>noErr</code>	0	No error
<code>nsvErr</code>	-35	Volume not found
<code>ioErr</code>	-36	I/O error
<code>fnfErr</code>	-43	File not found
<code>fLckdErr</code>	-45	File is locked
<code>vLckdErr</code>	-46	Volume is locked or read-only
<code>paramErr</code>	-50	Function not supported by volume
<code>volOfflineErr</code>	-53	Volume is offline
<code>wrgVolTypeErr</code>	-123	Not an HFS volume
<code>diffVolErr</code>	-1303	Files on different volumes
<code>afpAccessDenied</code>	-5000	User does not have the correct access
<code>afpObjectTypeErr</code>	-5025	Object is a directory, not a file
<code>afpSameObjectErr</code>	-5038	Source and destination files are the same

Creating File System Specifications

The `FSMakeFSSpec` function allows you to create `FSSpec` records.

FSMakeFSSpec

You can use the `FSMakeFSSpec` function to initialize an `FSSpec` record to particular values for a file or directory.

```
FUNCTION FSMakeFSSpec (vRefNum: Integer; dirID: LongInt;
                      fileName: Str255; VAR spec: FSSpec):
                      OSErr;
```

<code>vRefNum</code>	A volume specification. This parameter can contain a volume reference number, a working directory reference number, a drive number, or 0 (to specify the default volume).
<code>dirID</code>	A directory specification. This parameter usually specifies the parent directory ID of the target object. If the directory is sufficiently specified by either the <code>vRefNum</code> or <code>fileName</code> parameter, <code>dirID</code> can be set to 0. If you explicitly specify <code>dirID</code> (that is, if it has any value other than 0), and if <code>vRefNum</code> specifies a working directory reference number, <code>dirID</code> overrides the directory ID included in <code>vRefNum</code> . If the <code>fileName</code> parameter contains an empty string, <code>FSMakeFSSpec</code> creates an <code>FSSpec</code> record for a directory specified by either the <code>dirID</code> or <code>vRefNum</code> parameter.
<code>fileName</code>	A full or partial pathname. If <code>fileName</code> specifies a full pathname, <code>FSMakeFSSpec</code> ignores both the <code>vRefNum</code> and <code>dirID</code> parameters. A partial pathname might identify only the final target, or it might include one or more parent directory names. If <code>fileName</code> specifies a partial pathname, then <code>vRefNum</code> , <code>dirID</code> , or both must be valid.
<code>spec</code>	A file system specification to be filled in by <code>FSMakeFSSpec</code> .

DESCRIPTION

The `FSMakeFSSpec` function fills in the fields of the `spec` parameter using the information contained in the other three parameters. Call `FSMakeFSSpec` whenever you want to create an `FSSpec` record.

You can pass the input to `FSMakeFSSpec` in several ways. The chapter “File Manager” in this book explains how `FSMakeFSSpec` interprets its input.

If the specified volume is mounted and the specified parent directory exists, but the target file or directory doesn’t exist in that location, `FSMakeFSSpec` fills in the record and then returns `fnfErr` instead of `noErr`. The record is valid, but it describes a target that doesn’t exist. You can use the record for other operations, such as creating a file with the `FSpCreate` function.

In addition to the result codes that follow, `FSMakeFSSpec` can return a number of other File Manager error codes. If your application receives any result code other than `noErr` or `fnfErr`, all fields of the resulting `FSSpec` record are set to 0.

RESULT CODES

<code>noErr</code>	0	No error
<code>nsvErr</code>	-35	Volume doesn't exist
<code>fnfErr</code>	-43	File or directory does not exist (<code>FSSpec</code> is still valid)

Volume Access Routines

This section describes the high-level volume access routines. Unless your application has very specialized needs, you should be able to manage all volume access using the routines described in this section. In fact, most applications are likely to need only the `FlushVol` function described in the next section, "Updating Volumes."

When you call one of these routines, you specify a volume by a volume reference number (which you can obtain, for example, by calling the `GetVInfo` function, or from the reply record returned by the Standard File Package). You can also specify a volume by name, but this is generally discouraged, because there is no guarantee that volume names are unique.

Updating Volumes

When you close a file, you should call `FlushVol` to ensure that any changed contents of the file are written to the volume.

FlushVol

You can use the `FlushVol` function to write the contents of the volume buffer and update information about the volume.

```
FUNCTION FlushVol (volName: StringPtr; vRefNum: Integer): OSErr;
```

<code>volName</code>	A pointer to the name of a mounted volume.
<code>vRefNum</code>	A volume reference number, a working directory reference number, a drive number, or 0 for the default volume.

DESCRIPTION

On the specified volume, the `FlushVol` function writes the contents of the associated volume buffer and descriptive information about the volume (if they've changed since the last time `FlushVol` was called). This information is written to the volume.

RESULT CODES

noErr	0	No error
nsvErr	-35	No such volume
ioErr	-36	I/O error
bdNamErr	-37	Bad volume name
paramErr	-50	No default volume
nsDrvErr	-56	No such drive

Obtaining Volume Information

You can get information about a volume by calling the `GetVInfo` or `GetVRefNum` function.

GetVInfo

You can use the `GetVInfo` function to get information about a mounted volume.

```
FUNCTION GetVInfo (drvNum: Integer; volName: StringPtr;
                  VAR vRefNum: Integer;
                  VAR freeBytes: LongInt): OSErr;
```

<code>drvNum</code>	The drive number of the volume for which information is requested.
<code>volName</code>	On output, a pointer to the name of the specified volume.
<code>vRefNum</code>	The volume reference number of the specified volume.
<code>freeBytes</code>	The available space (in bytes) on the specified volume.

DESCRIPTION

The `GetVInfo` function returns the name, volume reference number, and available space (in bytes) for the specified volume. You specify a volume by providing its drive number in the `drvNum` parameter. You can pass 0 in the `drvNum` parameter to get information about the default volume.

RESULT CODES

noErr	0	No error
nsvErr	-35	No such volume
paramErr	-50	No default volume

GetVRefNum

You can use the `GetVRefNum` function to get a volume reference number from a file reference number.

```
FUNCTION GetVRefNum (refNum: Integer; VAR vRefNum: Integer):
    OSErr;
```

`refNum` The file reference number of an open file.

`vRefNum` On exit, the volume reference number of the volume containing the file specified by `refNum`.

DESCRIPTION

The `GetVRefNum` function returns the volume reference number of the volume containing the specified file. If you also want to determine the directory ID of the specified file's parent directory, call the `PBGetFCBInfo` function.

RESULT CODES

<code>noErr</code>	0	No error
<code>rfNumErr</code>	-51	Bad reference number

Application Launch File Routines

You can call `GetAppParms` to determine your application's name and the reference number of its resource file. When your application starts up, you can call `CountAppFiles` to determine whether the user selected any documents to open or print. If so, you can call `GetAppFiles` and `ClrAppFiles` to process the information passed to your application by the Finder.

Note

If your application supports high-level events, you receive this information from the Finder in an Open Documents or Print Documents event. ♦

GetAppParms

You can use the `GetAppParms` procedure to get information about the current application and about files selected by the user for opening or printing.

```
PROCEDURE GetAppParms(VAR apName: Str255; VAR apRefNum: Integer;
                     VAR apParam: Handle);
```

<code>apName</code>	On output, the name of the calling application.
<code>apRefNum</code>	On output, the reference number of the application's resource file.
<code>apParam</code>	On output, a handle to the Finder information about files to open or print.

DESCRIPTION

The `GetAppParms` procedure returns information about the current application. You can call `GetAppParms` at application launch time to determine which files, if any, the user has selected in the Finder for opening or printing. You can call `GetAppParms` at any time to determine the current application's name and the reference number of the application's resource fork.

The `GetAppParms` procedure returns the application's name in the `apName` parameter and the reference number of its resource fork in the `apRefNum` parameter. A handle to the Finder information is returned in `apParam`. This information consists of a word that encodes the message or action to be performed, a word that indicates how many files to process, and a list of Finder information about each such file. The Finder information has the structure of an `AppFile` record, except that the filename occupies only as many bytes as are required to hold the name (padded to an even number of bytes, if necessary). In general, it is easier to use the `GetAppFiles` procedure to access the Finder information.

SPECIAL CONSIDERATIONS

If you simply want to determine the application's resource file reference number, you can call the Resource Manager function `CurResFile` when your application starts up.

If you need more extensive information about the application than `GetAppParms` provides, you can use the Process Manager function `GetCurrentProcess`.

ASSEMBLY-LANGUAGE INFORMATION

You can get the application's name, reference number, and handle to the Finder information directly from the global variables `CurApName`, `CurApRefNum`, and `AppParmHandle`.

CountAppFiles

You can use the `CountAppFiles` procedure to determine how many documents (if any) the user has selected at application launch time for opening or printing.

```
PROCEDURE CountAppFiles (VAR message: Integer;
                        VAR count: Integer);
```

`message` The action to be performed on the selected files.
`count` The number of files selected.

DESCRIPTION

The `CountAppFiles` procedure deciphers the Finder information passed to your application and returns information about the files that were selected when your application was started up. On exit, the `count` parameter contains the number of selected files, and the `message` parameter contains an integer that indicates whether the files are to be opened or printed. The `message` parameter contains one of these constants:

```
CONST
    appOpen  =  0;    {open the document(s)}
    appPrint =  1;    {print the document(s)}
```

GetAppFiles

You can use the `GetAppFiles` procedure to retrieve information about each file selected at application startup for opening or printing.

```
PROCEDURE GetAppFiles (index: Integer; VAR theFile: AppFile);
```

`index` The index of the file whose information is returned.
`theFile` A structure containing the returned information.

DESCRIPTION

The `GetAppFiles` procedure returns information about a file that was selected when your application was started up (as listed in the Finder information). The `index` parameter indicates the file for which information should be returned; it must be between 1 and the number returned by `CountAppFiles`, inclusive.

ClrAppFiles

You can use the `ClrAppFiles` procedure to notify the Finder that you have processed the information about a file selected for opening or printing at application startup.

```
PROCEDURE ClrAppFiles (index: Integer);
```

`index` The index of the file whose information is to be cleared.

DESCRIPTION

The `ClrAppFiles` procedure changes the Finder information passed to your application about the specified file so that the Finder knows you've processed the file. The `index` parameter must be between 1 and the number returned by `CountAppFiles`, inclusive. You should call `ClrAppFiles` for every document your application opens or prints, so that the information returned by `CountAppFiles` and `GetAppFiles` is always correct. The `ClrAppFiles` procedure sets the file type in the Finder information to 0.

Result Codes

<code>noErr</code>	0	No error
<code>dirFulErr</code>	-33	File directory full
<code>dskFulErr</code>	-34	All allocation blocks on the volume are full
<code>nsvErr</code>	-35	Volume not found
<code>ioErr</code>	-36	I/O error
<code>bdNamErr</code>	-37	Bad filename or volume name
<code>fnOpnErr</code>	-38	File not open
<code>eofErr</code>	-39	Logical end-of-file reached
<code>posErr</code>	-40	Attempt to position mark before start of file
<code>tmfoErr</code>	-42	Too many files open
<code>fnfErr</code>	-43	File not found
<code>wPrErr</code>	-44	Hardware volume lock
<code>fLckdErr</code>	-45	File locked
<code>vLckdErr</code>	-46	Software volume lock
<code>fBsyErr</code>	-47	File is busy; one or more files are open; directory not empty or working directory control block is open
<code>dupFNErr</code>	-48	A file with the specified name and version number already exists
<code>opWrErr</code>	-49	File already open for writing
<code>paramErr</code>	-50	Parameter error
<code>rfNumErr</code>	-51	Reference number specifies nonexistent access path
<code>gfpErr</code>	-52	Error during <code>GetFPos</code>
<code>volOfflinErr</code>	-53	Volume is offline
<code>permErr</code>	-54	Attempt to open locked file for writing
<code>nsDrvErr</code>	-56	Specified drive number doesn't match any number in the drive queue
<code>wrPermErr</code>	-61	Read/write permission doesn't allow writing
<code>dirNFErr</code>	-120	Directory not found or incomplete pathname

Introduction to File Management

wrgVolTypeErr	-123	Not an HFS volume
notAFileErr	-1302	Specified file is a directory
diffVolErr	-1303	Files are on different volumes
sameFileErr	-1306	Source and destination files are the same
afpAccessDenied	-5000	User does not have the correct access to the file
afpObjectTypeErr	-5025	Object is a directory, not a file; a directory exists with that name
afpSameObjectErr	-5038	Source and destination files are the same

Introduction to File Management

Standard File Package

This chapter describes how your application can use the Standard File Package to manage the user interface for naming and identifying files. The Standard File Package displays the dialog boxes that let the user specify the names and locations of files to be saved or opened, and it reports the user's choices to your application.

The Standard File Package supports both standard and customized dialog boxes. The standard dialog boxes are sufficient for applications that do not require additional controls or other elements in the user interface. The chapter “Introduction to File Management” earlier in this book provides a detailed description of how to display the standard dialog boxes by calling two of the enhanced Standard File Package routines introduced in system software version 7.0. You need to read this chapter if your application needs to use features not described in that earlier chapter (such as customized dialog boxes or a special file filter function). You also need to read this chapter if you want your application to run in an environment where the new routines are not available and your development system does not provide glue code that allows you to call the enhanced routines in earlier system software versions.

To use this chapter, you should be familiar with the Dialog Manager, the Control Manager, and the Finder. You need to know about the Dialog Manager if you want to provide a modal-dialog filter function that handles events received from the Event Manager before they are passed to the `ModalDialog` procedure (which the Standard File Package uses to manage both standard and customized dialog boxes). You need to know about the Control Manager if you want to customize the user interface by adding controls (such as radio buttons or pop-up menus). You need to know about the Finder if your application supports stationery documents. See the appropriate chapters in *Inside Macintosh: Macintosh Toolbox Essentials* for specific information about these system software components.

This chapter provides an introduction to the Standard File Package and then discusses

- how you can display the standard file selection dialog boxes
- how the Standard File Package interprets user actions in those dialog boxes
- how to manage customized dialog boxes

Standard File Package

- how to set the directory whose contents are listed in a dialog box
- how to allow the user to select a volume or directory
- how to use the original Standard File Package routines

About the Standard File Package

Macintosh applications typically have a File menu from which the user can save and open documents, via the Save, Save As, and Open commands. When the user chooses Open to open an existing document, your application needs to determine which document to open. Similarly, when the user chooses Save As, or Save when the document is untitled, your application needs to ask the user for the name and location of the file in which the document is to be saved.

The Standard File Package provides a number of routines that handle the user interface between the user and your application when the user saves or opens a document. It displays dialog boxes through which the user specifies the name and location of the document to be saved or opened. It also allows your application to customize the dialog boxes and, through callback routines, to handle user actions during the dialogs. The Standard File Package procedures return information about the user's choices to your application through a reply record.

The Standard File Package is available in all versions of system software. However, significant improvements were made to the package in system software version 7.0. The Standard File Package in version 7.0 introduces

- a pair of simplified procedures (`StandardGetFile` and `StandardPutFile`) that you call to display and handle the standard Open and Save dialog boxes
- a pair of customizable procedures (`CustomGetFile` and `CustomPutFile`) that you call when you need more control over the interaction
- a new reply record (`StandardFileReply`) that identifies files and folders with a file system specification record and that accommodates the new Finder features introduced in system software version 7.0
- a new layout for the standard dialog boxes

This section describes in detail the standard and customized user interfaces provided by the enhanced Standard File Package in system software version 7.0 and later. If your application is to run in earlier system software versions as well, you should read the section “Using the Original Procedures” on page 4-215.

IMPORTANT

If you use the enhanced routines introduced in system software version 7.0, you must also support the Open Documents Apple event. ▲

Standard User Interfaces

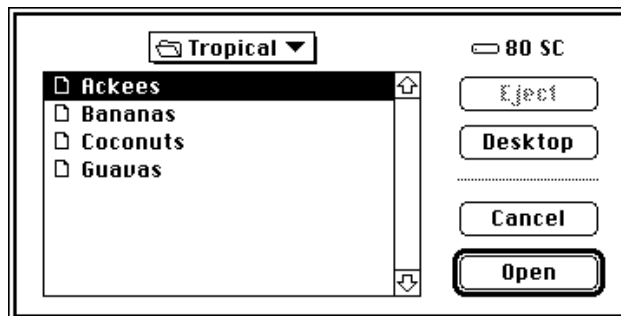
If your application has no special interface requirements, you can use the `StandardGetFile` and `StandardPutFile` procedures to display the standard dialog boxes for opening and saving documents.

Opening Files

You use the `StandardGetFile` procedure when you want to let the user select a file to be opened. Figure 4-1 illustrates a sample dialog box displayed by `StandardGetFile`.

The directory whose contents are listed in the **display list** in the dialog box displayed by `StandardGetFile` is known as the **current directory**. In Figure 4-1, the current directory is named “Tropical.” The user can change the current directory in several ways. To ascend the directory hierarchy from the current directory, the user can click the directory pop-up menu and select a new directory from among those in the menu. To ascend one level of the directory hierarchy, the user can click the volume icon. To ascend immediately to the top of the directory hierarchy, the user can click the Desktop button.

Figure 4-1 The default Open dialog box



To descend the directory hierarchy, the user can double-click any of the folder names in the list (or select a folder by clicking its name once and then clicking the Open button). Whenever the current directory changes, the list of folders and files is updated to reflect the contents of the new current directory.

The volume on which the current directory is located is the **current volume** (or **current disk**), whose name is displayed to the right of the directory pop-up menu. If the current volume is a removable volume, the Eject button is active. The user can click Eject to eject the current volume and insert another, which then becomes the current volume. If the user inserts an uninitialized or otherwise unreadable disk, the Standard File Package calls the Disk Initialization Manager to provide the standard user interface for initializing and naming a disk. See the chapter “Disk Initialization Manager” in this book for details.

Note that the list of files and folders always contains all folders in the current directory, but it might not contain all files in the current directory. When you call

Standard File Package

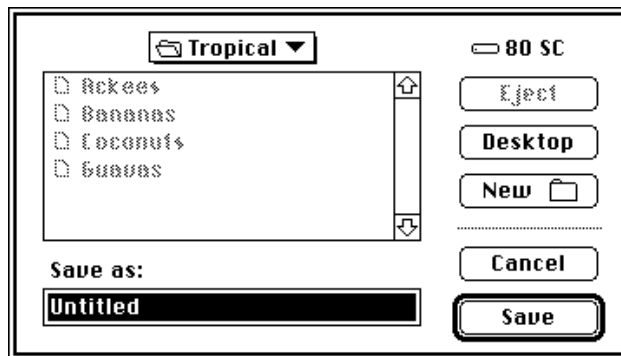
`StandardGetFile`, you can supply a list of the file types that your application can open. The `StandardGetFile` procedure then displays only files of the specified types. You can also supply your own file filter function to help determine which files are displayed. (See “Writing a File Filter Function” on page 4-195 for details.)

When the user is opening a document, `StandardGetFile` interprets some keystrokes as selectors in the displayed list. If the user presses A, for example, `StandardGetFile` selects the first item in the list that starts with the letter a (or, if no items in the list start with the letter a, the item that starts with the letter closest to a). The Standard File Package sets a timer on keystrokes: keystrokes in rapid succession form a string; keystrokes spaced in time are processed separately. See “Keyboard Equivalents” on page 4-182 for a complete list of keyboard equivalents recognized by `StandardGetFile`.

Saving Files

You use the `StandardPutFile` procedure when you want to let the user specify a name and location for a file to be saved. Figure 4-2 illustrates a sample dialog box displayed by `StandardPutFile`.

Figure 4-2 The default Save dialog box

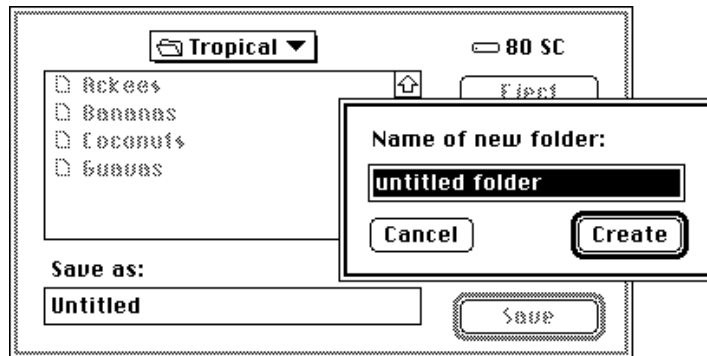


The dialog box displayed by `StandardPutFile` is similar to that displayed by `StandardGetFile`, but includes three additional items. The Save dialog box includes a filename field in which the user can type the name under which to save the file. This filename field is a `TextEdit` field that permits all the standard editing operations (cut, copy, paste, and so forth). Above the filename field is a line of text specified by your application.

When the user is saving a document, `StandardPutFile` can direct keystrokes to either of two targets: the filename field or the displayed list. When the dialog box first appears, keystrokes are directed to the filename field. If the user presses the Tab key or clicks to select an item in the displayed list, subsequent keystrokes are interpreted as selectors in the displayed list. Each time the user presses the Tab key, keyboard input shifts between the two targets.

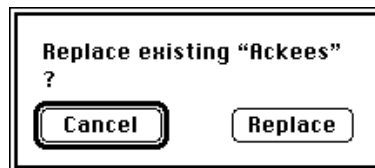
The third additional item in the Save dialog is the New Folder button. When the user clicks the New Folder button, the Standard File Package presents a subsidiary dialog box like the one shown in Figure 4-3.

Figure 4-3 The New Folder dialog box



If the user asks to save a file under a name that already exists at the specified location, the Standard File Package displays a subsidiary dialog box to verify that the new file should replace the existing file, as illustrated in Figure 4-4.

Figure 4-4 The name conflict dialog box



The `StandardGetFile` and `StandardPutFile` procedures always display the new dialog boxes. The procedures available before version 7.0 (`SFGetFile`, `SFPutFile`, `SFPGGetFile`, and `SFPPutFile`) also display the new dialog boxes when running in version 7.0, unless your application has customized the dialog box. For more details on how the version 7.0 Standard File Package handles earlier procedures, see “Using the Original Procedures” on page 4-215.

Keyboard Equivalents

The Standard File Package recognizes a long list of keyboard equivalents during dialogs.

Keystrokes	Action
Up Arrow	Scroll up (backward) through displayed list
Down Arrow	Scroll down (forward) through displayed list
Command-Up Arrow	Display contents of parent directory
Command-Down Arrow	Display contents of selected directory or volume
Command-Left Arrow	Display contents of previous volume
Command-Right Arrow	Display contents of next volume
Command-Shift-Up Arrow	Display contents of desktop
Command-Shift-1	Eject disk in drive 1
Command-Shift-2	Eject disk in drive 2
Tab	Move to next keyboard target
Return <i>or</i> Enter	Invoke the default option for the dialog box (Open or Save)
Escape <i>or</i> Command-.	Cancel
Command-O	Open the selected item
Command-D	Display contents of desktop
Command-N	Create a new folder
Option-Command-O <i>or</i> Option-[click Open]	Select the target of the selected alias item instead of opening it

When the user uses a keyboard equivalent to select a button in the dialog box, the button blinks.

Customized User Interfaces

The standard user interfaces provided by the `StandardGetFile` and `StandardPutFile` procedures might not be adequate for the needs of certain applications. To handle such cases, the Standard File Package provides several routines that you can use to present a customized user interface when opening or saving files. This section gives some simple examples of how you might want to customize the user interfaces and suggests some guidelines you should follow when doing so.

IMPORTANT

You should alter the standard user interfaces only if necessary. Apple Computer, Inc., does not guarantee future compatibility for your application if you use a customized dialog box. ▲

Saving Files

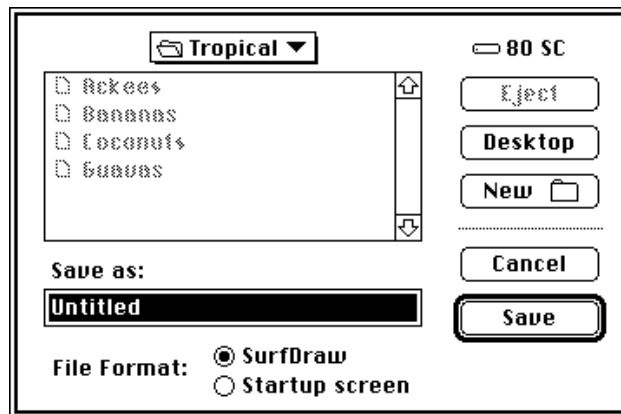
Perhaps the most common reason to customize one of the Standard File Package dialog boxes is to allow the user to save a document in one of several file formats supported by the

application. For example, a word-processing application might let the user save a document in the application's own format, in an interchange format, as a file of type 'TEXT', and so on.

It is usually best to allow the user to select a file format from within the dialog box displayed in response to a Save or Save As menu command. To do this, you need to add items to the standard dialog box and process user actions in those new items.

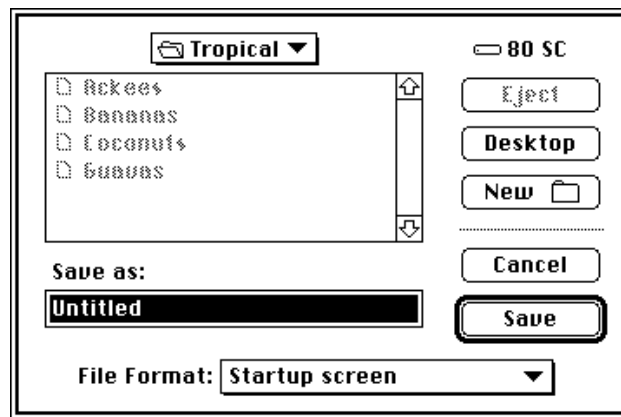
If your application supports only a few file formats, you could simply add the required number of radio buttons to the standard dialog box, as illustrated in Figure 4-5. The application presenting this dialog box supports only two file formats, its own proprietary format (SurfDraw) and the format used for startup screens.

Figure 4-5 The Save dialog box customized with radio buttons



If your application supports more than a couple of alternate file formats, you could add a pop-up menu, as shown in Figure 4-6.

Figure 4-6 The Save dialog box customized with a pop-up menu

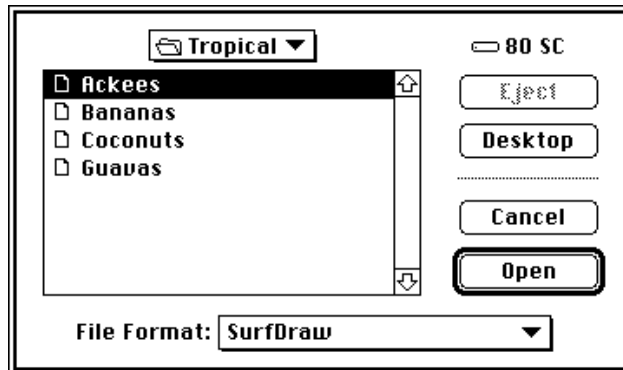


Opening Files

Your application might also allow the user to open a number of different types of files. In this case, there is less need to customize the Open dialog box than the Save dialog box because you can simply list all the kinds of files your application supports. To avoid clutter in the list of files and folders, however, you might wish to filter out all but one of those types. In this way, the user can dynamically select which type of file to view in the list.

Once again, you might accomplish this by adding radio buttons or a pop-up menu to the Open dialog box, depending on the number of different file types your application supports. Figure 4-7 illustrates a customized Open dialog box that contains a pop-up menu. Only files of the indicated type (and, of course, folders) appear in the list of items available to open.

Figure 4-7 The Open dialog box customized with a pop-up menu



For details on some techniques you can use to add items to the standard user interface and process user actions with those additional items, see “Customizing the User Interface” on page 4-191. Note in particular that Listing 4-3, Listing 4-8, and Listing 4-9 together provide a fairly complete implementation of the pop-up menu illustrated in Figure 4-7.

Note

Remember that the user might also open one of your application’s documents from the Finder (by double-clicking its icon, for example). As a result, you should in general avoid customizing the Open dialog box for files. ♦

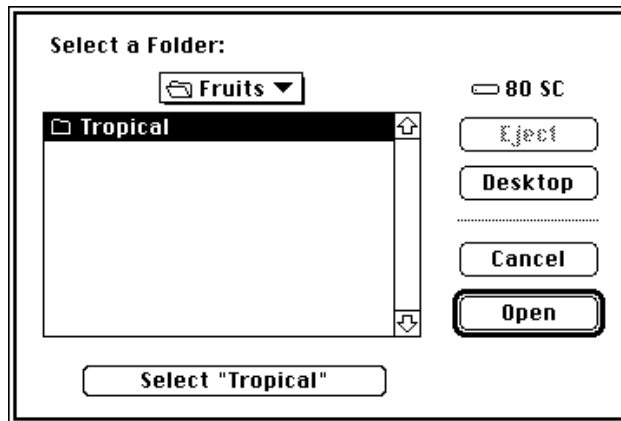
Selecting Volumes and Directories

Sometimes you need to allow the user to select a directory or a volume, not a file. For example, the user might want to select a directory as a first step in searching all the files in the directory for some important information. Similarly, the user might need to select a volume before backing up all the files on that volume.

The standard Open dialog box, however, is designed for selecting files, not volumes or directories. When the user selects a volume or directory from the items in the displayed list and clicks the Open button, the volume or directory is opened and its contents are displayed in the list. The standard Open dialog boxes provide no obvious mechanism for choosing a selected directory instead of opening it.

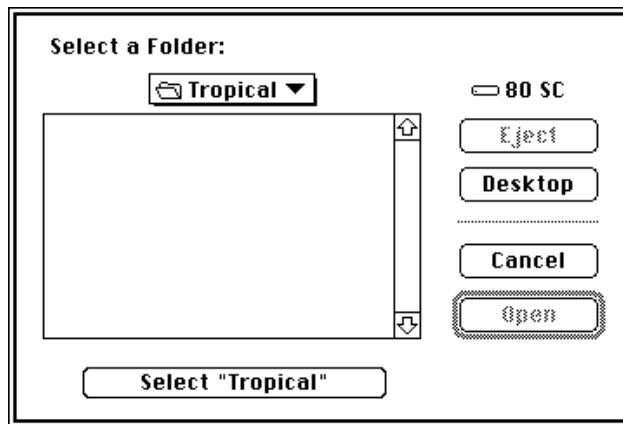
To allow the user to select a directory—including the volume's root directory, the volume itself—you can add an additional button to the standard Open dialog box. By clicking this button, the user can select a highlighted directory, not open it. This button gives the user an obvious way to select a directory while preserving the well-known mechanism for opening directories to search for the desired directory. Figure 4-8 illustrates the standard Open dialog box modified to include a Select button and a prompt informing the user of the type of action required.

Figure 4-8 The Open dialog box customized to allow selection of a directory

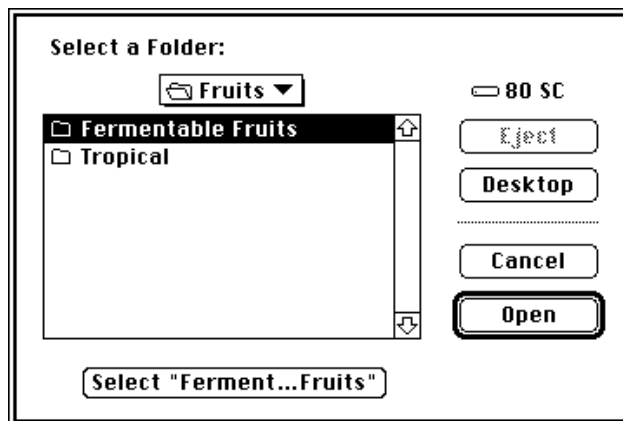


The Select button should display the name of the directory that is selected if the user clicks the button. This, together with the prompt displayed at the top of the dialog box, helps the user differentiate this directory selection dialog box from the standard file opening dialog box. All the other items in the dialog box should maintain their standard appearance and behavior. Any existing keyboard equivalents (in particular, the use of Return and Enter to select the default button) should be preserved. Command-S is recommended as a keyboard equivalent for the new Select button, paralleling the use of Command-D to select the Desktop button and Command-O to select the Open button.

To help maintain consistency among applications using this scheme for selecting directories, your application should open the folder displayed in the pop-up menu if there is no selected item and the user clicks the Select button. In addition, you should disable the Open button if no directory is currently selected. Figure 4-9 illustrates the recommended appearance of the directory selection dialog box in this case.

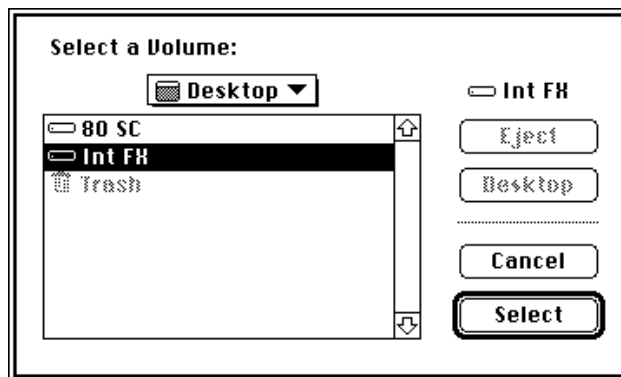
Figure 4-9 The Open dialog box when no directory is selected

If the name of the directory is too long to fit in the Select button, you should abbreviate the name using an ellipsis character, as shown in Figure 4-10.

Figure 4-10 The Open dialog box with a long directory name abbreviated

See “Selecting a Directory” beginning on page 4-209 for details on how you can create and manage a directory selection dialog box.

The directory selection dialog boxes illustrated here allow the user to specify the root directory in a volume, which effectively selects the volume itself. However, you might want to limit the user’s selections to the available volumes. To do that, you can create a volume selection dialog box, shown in Figure 4-11.

Figure 4-11 A volume selection dialog box

Notice that the volume selection dialog box uses a prompt specific to selecting a volume and that the Open button is now a Select button. There is no need for a separate Select button, because the user should not be allowed to open any of the listed volumes. (For this same reason, the pop-up menu should not pop up if clicked.) See “Selecting a Volume” on page 4-213 for instructions on implementing a volume selection dialog box.

User Interface Guidelines

In general, you should customize the user interface only if necessary. If you do modify the standard dialog boxes presented by the Standard File Package, you should keep these user interface guidelines in mind:

- Customize a dialog box only by adding items to the standard dialog boxes. Avoid removing existing items from the standard boxes or altering the operation of existing items. In particular, you should avoid modifying the keyboard equivalents recognized by the Standard File Package.
- Add only those items that are necessary for your application to complete the requested action successfully. Avoid adding items that provide unnecessary information or items that provide no information at all (such as logos, icons, or other “window-dressing”).
- Whenever possible, use controls such as radio buttons or pop-up menus whose effects are visible within the dialog box itself. Avoid controls whose use calls subsidiary modal dialog boxes that the user must dismiss before continuing.
- Use controls or other items that are already familiar to the user. Avoid using customized controls that are not also used elsewhere in your application.

Your overriding concern should be to make the customized file identification dialog boxes in your application as similar to the standard dialog boxes as possible while providing the additional capabilities you need.

Using the Standard File Package

You use the Standard File Package to handle the user interface when the user must specify a file to be saved or opened. You typically call the Standard File Package after the user chooses Save, Save As, or Open from the File menu.

When saving a document, you call one of the `PutFile` procedures; when opening a document, you call one of the `GetFile` procedures. The Standard File Package in version 7.0 introduces two pairs of enhanced procedures:

- `StandardPutFile` and `StandardGetFile`, for presenting the standard interface
- `CustomPutFile` and `CustomGetFile`, for presenting a customized interface

Before calling the enhanced Standard File Package procedures, verify that they are available by calling the `Gestalt` function with the `gestaltStandardFileAttr` selector. If `Gestalt` sets the `gestaltStandardFile58` bit in the reply, the four enhanced procedures are available.

If the enhanced procedures are not available, you need to use the original Standard File Package procedures that are available in all system software versions:

- `SFPutFile` and `SFGetFile`, for presenting the standard interface
- `SFPPutFile` and `SFPGetFile`, for presenting a customized interface

This section focuses on the enhanced procedures introduced in system software version 7.0. If you need to use the original procedures, see “Using the Original Procedures” on page 4-215. You can adapt most of the techniques shown in this section for use with the original procedures. In general, however, the original procedures are slightly harder to use and somewhat less powerful than their enhanced counterparts.

All the enhanced procedures return the results of the dialog boxes in a new reply record, `StandardFileReply`.

```
TYPE StandardFileReply =
    RECORD
        sfGood:           Boolean;      {TRUE if user did not cancel}
        sfReplacing:     Boolean;      {TRUE if replacing file with same name}
        sfType:          OSType;       {file type}
        sfFile:          FSSpec;       {selected file, folder, or volume}
        sfScript:        ScriptCode;   {script of file, folder, or volume name}
        sfFlags:         Integer;      {Finder flags of selected item}
        sfIsFolder:      Boolean;      {selected item is a folder}
        sfIsVolume:      Boolean;      {selected item is a volume}
        sfReserved1:     LongInt;      {reserved}
        sfReserved2:     Integer;      {reserved}
    END;
```

The reply record identifies selected files with a file system specification (`FSSpec`) record. You can pass the `FSSpec` record directly to the File Manager functions that recognize

Standard File Package

FSSpec records, such as FSpOpenDF or FSpCreate. The reply record also contains additional fields that support the Finder features introduced in system software version 7.0.

The `sfGood` field reports whether the reply record is valid—that is, whether your application can use the information in the other fields. The field is set to `TRUE` after the user clicks Save or Open, and to `FALSE` after the user clicks Cancel.

Your application needs to look primarily at the `sfFile` and `sfReplacing` fields when the `sfGood` field contains `TRUE`. The `sfFile` field contains a file system specification record that describes the selected file or folder. If the selected file is a stationery pad, the reply record describes the file itself, not a copy of the file.

The `sfReplacing` field reports whether a file to be saved replaces an existing file of the same name. This field is valid only after a call to the `StandardPutFile` or `CustomPutFile` procedure. Your application can rely on the value of this field instead of checking for and handling name conflicts itself.

Note

See “Enhanced Standard File Reply Record” on page 4-217 for a complete description of the fields of the `StandardFileReply` record. ♦

The Standard File Package fills in the reply record and returns when the user completes one of its dialog boxes—either by selecting a file and clicking Save or Open, or by clicking Cancel. Your application checks the values in the reply record to see what action to take, if any. If the selected item is an alias for another item, the Standard File Package resolves the alias and places a file system specification record for the target in the `sfFile` field when the user completes the dialog box. (See the chapter “Finder Interface” of *Inside Macintosh: Macintosh Toolbox Essentials* for a description of aliases.)

Presenting the Standard User Interface

You can use the standard dialog boxes provided by the Standard File Package to prompt the user for the name of a file to open or a filename and location to use when saving a document. Use `StandardGetFile` to present the standard interface when opening a file and `StandardPutFile` to present the standard interface when saving a file.

Listing 4-1 illustrates how your application can use `StandardGetFile` to elicit a file specification after the user chooses Open from the File menu.

Listing 4-1 Handling the Open menu command

```
FUNCTION DoOpenCmd: OSErr;
VAR
    myReply:      StandardFileReply;    {Standard File reply record}
    myTypes:      SFTYPEList;           {types of files to display}
    myErr:        OSErr;
BEGIN
```

Standard File Package

```

myTypes[0] := 'TEXT';           {display text files only}
StandardGetFile(NIL, 1, myTypes, myReply);
IF myReply.sfGood THEN
    myErr := DoOpenFile(myReply.sfFile)
ELSE
    myErr := UsrCanceledErr;
DoOpenCmd := myErr;
END;

```

If the user dismisses the dialog box by clicking the Open button, the reply record field `myReply.sfGood` is set to `TRUE`; in that case, the function defined in Listing 4-1 calls the application-defined function `DoOpenFile`, passing it the file system specification record contained in the reply record. For a sample definition of the `DoOpenFile` function, see the chapter “Introduction to File Management” in this book.

The third parameter to `StandardGetFile` is a list of file types that are to appear in the list of files and folders; the second parameter is the number of items in that list of file types. The list of file types is of type `SFTypeList`.

```
TYPE SFTypeList = ARRAY[0..3] OF OSType;
```

If you need to display more than four types of files, you can define a new data type that is large enough to hold all the types you need. For example, you can define the data type `MyTypeList` to hold ten file types:

```
TYPE MyTypeList = ARRAY[0..9] OF OSType;
    MyTListPtr = ^MyTypeList;
```

Listing 4-2 shows how to call `StandardGetFile` using an expanded type list.

Listing 4-2 Specifying more than four file types

```

FUNCTION DoOpenCmd: OSerr;
VAR
    myReply: StandardFileReply; {Standard File reply record}
    myTypes: MyTypeList;        {types of files to display}
    myErr: OSerr;
BEGIN
    myTypes[0] := 'TEXT';       {first file type to display}
    {Put other assignments here.}
    myTypes[9] := 'RTFT';       {tenth file type to display}
    StandardGetFile(NIL, 1, MyTListPtr(myTypes)^, myReply);
    IF myReply.sfGood THEN
        myErr := DoOpenFile(myReply.sfFile)

    ELSE

```

Standard File Package

```

myErr := UsrCanceledErr;
DoOpenCmd := myErr;
END;

```

Note

To display all file types in the dialog box, pass -1 as the second parameter. Invisible files and folders are not shown in the dialog box unless you pass -1 in that parameter. If you pass -1 as the second parameter when calling `CustomGetFile`, the dialog box also lists folders; this is not true when you call `StandardGetFile`. ♦

The first parameter passed to `StandardGetFile` is the address of a file filter function, a function that helps determine which files appear in the list of files to open. (In Listing 4-1, this address is `NIL`, indicating that all files of the specified type are to be listed.) See “Writing a File Filter Function” on page 4-195 for details on defining a filter function for use with `StandardGetFile`.

Customizing the User Interface

If your application requires it, you can customize the user interface for identifying files. To customize a dialog box, you should

- design your dialog box and create the resources that describe it
- write callback routines, if necessary, to process user actions in the dialog box
- call the Standard File Package using the `CustomPutFile` and `CustomGetFile` procedures, passing the resource IDs of the customized dialog boxes and pointers to the callback routines

Depending on the level of customizing you require in your dialog box, you may need to write as many as four different callback routines:

- a file filter function for determining which files the user can open
- a dialog hook function for handling user actions in the dialog boxes
- a modal-dialog filter function for handling user events received from the Event Manager
- an activation procedure for highlighting the display when keyboard input is directed at a customized field defined by your application

To provide the interface illustrated in Figure 4-7, for example, you could replace the definition of `DoOpenCmd` given earlier in Listing 4-1 by the definition given in Listing 4-3.

In addition to the information passed to `StandardGetFile`, `CustomGetFile` requires the resource ID of the customized dialog box, the location of the dialog box on the screen, and pointers to any callback routines and private data you are using.

Listing 4-3 Presenting a customized Open dialog box

```

FUNCTION DoOpenCmd: OSErr;
VAR
    myReply:      StandardFileReply;    {Standard File reply record}
    myTypes:      SFTYPEList;           {types of files to display}
    myPoint:      Point;                {upper-left corner of box}
    myErr:        OSErr;
CONST
    kCustomGetDialog = 4000;            {resource ID of custom dialog}
BEGIN
    myErr := noErr;
    SetPt(myPoint, -1, -1);             {center the dialog}
    myTypes[0] := 'SRFD';               {SurfDraw files}
    myTypes[1] := 'STUP';               {startup screens}
    myTypes[2] := 'PICT';               {picture files}
    myTypes[3] := 'RTFT';               {rich text format}

    CustomGetFile(@MyCustomFileFilter, 4, myTypes, myReply,
                  kCustomGetDialog, myPoint, @MyDlgHook,
                  NIL, NIL, NIL, NIL);
    IF myReply.sfGood THEN
        myErr := DoOpenFile(myReply.sfFile);
    DoOpenCmd := myErr;
END;

```

In Listing 4-3, CustomGetFile is passed two callback routines, a file filter function (MyCustomFileFilter) and a dialog hook function (MyDlgHook). See Listing 4-8 (page 4-196) and Listing 4-9 (page 4-202) for sample definitions of these functions.

You can also supply data of your own to the callback routines through a new parameter, yourDataPtr, which you pass to CustomGetFile and CustomPutFile.

Customizing Dialog Boxes

To describe a dialog box, you supply a 'DLOG' resource that defines the box itself and a 'DITL' resource that defines the items in the dialog box.

Listing 4-4 shows the resource definition of the default Open dialog box, in Rez input format. (Rez is the resource compiler provided with Apple's Macintosh Programmer's Workshop [MPW]. For a description of Rez format, see the manual that accompanies the MPW software, *MPW: Macintosh Programmer's Development Environment*.)

Listing 4-4 The definition of the default Open dialog box

```
resource 'DLOG' (-6042, purgeable)
{
    {0, 0, 166, 344}, dBoxProc, invisible, noGoAway, 0,
    -6042, "", noAutoCenter
};
```

Listing 4-5 shows the resource definition of the default Save dialog box, in Rez input format.

Listing 4-5 The definition of the default Save dialog box

```
resource 'DLOG' (-6043, purgeable)
{
    {0, 0, 188, 344}, dBoxProc, invisible, noGoAway, 0,
    -6043, "", noAutoCenter
};
```

Note

You can also use the stand-alone resource editor ResEdit, available from Apple Computer, Inc., or other resource-editing utilities available from third-party developers to create customized dialog box and dialog item list resources. ♦

You must provide an item list (in a 'DITL' resource with the ID specified in the 'DLOG' resource) for each dialog box you define. Add new items to the end of the default lists. CustomGetFile expects the first 9 items in a customized dialog box to have the same functions as the corresponding items in the StandardGetFile dialog box; CustomPutFile expects the first 12 items to have the same functions as the corresponding items in the StandardPutFile dialog box. If you want to eliminate one of the standard items from the display, leave it in the item list but place its coordinates outside the bounds of the dialog box rectangle.

Listing 4-6 shows the dialog item list for the default Open dialog box, in Rez input format. See “Writing a Dialog Hook Function” beginning on page 4-196 for a list of the dialog box elements these items represent.

Listing 4-6 The item list for the default Open dialog box

```
resource 'DITL' (-6042)
{
    {
        {135, 252, 155, 332}, Button { enabled, "Open" },
        {104, 252, 124, 332}, Button { enabled, "Cancel" },
        {0, 0, 0, 0}, HelpItem { disabled, HMSCanhdlg {-6042}},
        {8, 235, 24, 337}, UserItem { enabled },
    }
};
```

Standard File Package

```

    {32, 252, 52, 332}, Button { enabled, "Eject" },
    {60, 252, 80, 332}, Button { enabled, "Desktop" },
    {29, 12, 159, 230}, UserItem { enabled },
    {6, 12, 25, 230}, UserItem { enabled },
    {91, 251, 92, 333}, Picture { disabled, 11 },
} };

```

Listing 4-7 shows the dialog item list for the default Save dialog box, in Rez input format.

Listing 4-7 The item list for the default Save dialog box

```

resource 'DITL'(-6043)
{
    {
        {161, 252, 181, 332}, Button { enabled, "Save" },
        {130, 252, 150, 332}, Button { enabled, "Cancel" },
        {0, 0, 0, 0}, HelpItem { disabled, HMScanhdlg {-6043}},
        {8, 235, 24, 337}, UserItem { enabled },
        {32, 252, 52, 332}, Button { enabled, "Eject" },
        {60, 252, 80, 332}, Button { enabled, "Desktop" },
        {29, 12, 127, 230}, UserItem { enabled },
        {6, 12, 25, 230}, UserItem { enabled },
        {119, 250, 120, 334}, Picture { disabled, 11 },
        {157, 15, 173, 227}, EditText { enabled, "" },
        {136, 15, 152, 227}, StaticText { disabled, "Save as:" },
        {88, 252, 108, 332}, UserItem { disabled },
    }
};

```

The third item in each list (`HelpItem`) supplies Apple's Balloon Help for items in the dialog box. This third item specifies the resource ID of the `'hdlg'` resource that contains the help strings for the standard dialog items. If you want to modify the help text of an existing dialog item, you should copy the original `'hdlg'` resource from the System file into your application's resource fork and modify the text in the copied resource as desired; then you must change the resource ID specified in `HelpItem` to the resource ID of the copied and modified resource. To provide Balloon Help for your own items, supply a second `'hdlg'` resource and reference it with another help item at the end of the list. The existing items retain their default text (unless you change that text, as described).

The default dialog item lists used by the original Standard File Package routines do not contain help items, but the Standard File Package does provide Balloon Help when you call those routines in system software version 7.0 and later. If you call one of the original routines and the specified dialog item list does not contain any help items, the Standard File Package uses its default help text for the standard dialog items. If, however, the dialog item list does contain a help item, the Standard File Package assumes that your application provides the text for all help items, including the standard dialog items.

Note

The default Standard File Package dialog boxes support color. The System file contains 'dctb' resources with the same resource IDs as the default dialog boxes, so that the Dialog Manager uses color graphics ports for the default dialog boxes. (See the chapter “Dialog Manager” of *Inside Macintosh: Macintosh Toolbox Essentials* for a description of the 'dctb' resource.) If you create your own dialog boxes, include 'dctb' resources. ♦

Writing a File Filter Function

A **file filter function** determines which files appear in the displayed list when the user is opening a file. Both `StandardGetFile` and `CustomGetFile` recognize file filter functions.

When the Standard File Package is displaying the contents of a volume or folder, it checks the file type of each file and filters out files whose types do not match your application's specifications. (Your application can specify which file types are to be displayed through the `typeList` parameter to either `StandardGetFile` or `CustomGetFile`, as described in “Presenting the Standard User Interface” beginning on page 4-189.) If your application also supplies a file filter function, the Standard File Package calls that function each time it identifies a file of an acceptable type.

The file filter function receives a pointer to the file's catalog information record (described in the chapter “File Manager” in this book). The function evaluates the catalog entry and returns a Boolean value that determines whether the file is filtered (that is, a value of `TRUE` suppresses display of the filename, and a value of `FALSE` allows the display). If you do not supply a file filter function, the Standard File Package displays all files of the specified types.

A file filter function to be called by `StandardGetFile` must use this syntax:

```
FUNCTION MyStandardFileFilter (pb: CInfoBPtr): Boolean;
```

The single parameter passed to your standard file filter function is the address of a catalog information parameter block; see the chapter “File Manager” in this book for a description of the fields of that parameter block.

When `CustomGetFile` calls your file filter function, it can also receive a pointer to any data that you passed in through the call to `CustomGetFile`. A file filter function to be called by `CustomGetFile` must use this syntax:

```
FUNCTION MyCustomFileFilter (pb: CInfoBPtr; myDataPtr: Ptr):  
    Boolean;
```

Listing 4-8 shows a sample file filter function to be called by `CustomGetFile`. You might define a file filter function like this to support the custom dialog box illustrated in Figure 4-7, which lists files of the type shown in the pop-up box.

Listing 4-8 A sample file filter function

```

FUNCTION MyCustomFileFilter (pb: CInfoBPBPtr; myDataPtr: Ptr): Boolean;
BEGIN
    MyCustomFileFilter := TRUE;                {default: don't show the file}
    IF pb^.ioFlFndrInfo.fdType = gTypesArray[gCurrentType] THEN
        MyCustomFileFilter := FALSE;           {show the file}
END;

```

In Listing 4-8, the application global variable `gCurrentType` contains the index in the array `gTypesArray` of the currently selected file type. If the type of a file passed in for evaluation matches the current file type, the filter returns `FALSE`, indicating that `StandardGetFile` should put it in the list. See Listing 4-9 (page 4-202) for an example of how you can use a dialog hook function to change the value of `gCurrentType` according to user selections in the pop-up menu control.

Writing a Dialog Hook Function

A **dialog hook function** handles item selections in a dialog box. It receives a pointer to the dialog record and an item number from the `ModalDialog` procedure via the Standard File Package each time the user selects one of the dialog items. Your dialog hook function checks the item number of each selected item, and then either handles the selection or passes it back to the Standard File Package.

If you provide a dialog hook function, `CustomPutFile` and `CustomGetFile` call your function immediately after calling `ModalDialog`. They pass your function the item number returned by `ModalDialog`, a pointer to the dialog record, and a pointer to the data received from your application, if any. The dialog hook function must use this syntax:

```

FUNCTION MyDlgHook (item: Integer; theDialog: DialogPtr;
                    myDataPtr: Ptr): Integer;

```

Your dialog hook function returns as its function result an integer that is either the item number passed to it or some other item number. If it returns one of the item numbers in the following list of constants, the Standard File Package handles the selected item as described later in this section. If your dialog hook function does not handle a selection, it should pass the item number back to the Standard File Package for processing by setting its return value equal to the item number.

```

CONST {items that appear in both the Open and Save dialog boxes}
    sfItemOpenButton      = 1;        {Save or Open button}
    sfItemCancelButton    = 2;        {Cancel button}
    sfItemBalloonHelp     = 3;        {Balloon Help}
    sfItemVolumeUser      = 4;        {volume icon and name}
    sfItemEjectButton     = 5;        {Eject button}
    sfItemDesktopButton   = 6;        {Desktop button}

```

Standard File Package

```

sfItemFileListUser      = 7;      {display list}
sfItemPopUpMenuUser     = 8;      {directory pop-up menu}
sfItemDividerLinePict   = 9;      {dividing line between buttons}

{items that appear in Save dialog boxes only}
sfItemFileNameTextEdit  = 10;     {filename field}
sfItemPromptStaticText  = 11;     {filename prompt text area}
sfItemNewFolderUser     = 12;     {New Folder button}

```

You must write your own dialog hook function to handle any items you have added to the dialog box.

Note

The constants that represent disabled items (`sfItemBalloonHelp`, `sfItemDividerLinePict`, and `sfItemPromptStaticText`) have no effect, but they are defined in the header files for the sake of completeness. ♦

The Standard File Package also recognizes a number of constants that do not represent any actual item in the dialog list; these constants are known as **pseudo-items**. There are two kinds of pseudo-items:

- pseudo-items passed to your dialog hook function by the Standard File Package
- pseudo-items passed to the Standard File Package by your dialog hook function

The `sfHookFirstCall` constant is an example of the first kind of pseudo-item. The Standard File Package sends this pseudo-item to your dialog hook function immediately before it displays the dialog box. Your function typically reacts to this item number by performing any necessary initialization.

You can pass back other pseudo-items to indicate that you've handled the user selection or to request some action by the Standard File Package. For example, if the list of files and folders must be rebuilt because of a user selection, you can pass back the pseudo-item `sfHookRebuildList`. Similarly, when your application handles the selection and needs no further action by the Standard File Package, it should return `sfHookNullEvent`. When the dialog hook function passes either `sfHookNullEvent` or an item number that the Standard File Package doesn't recognize, it does nothing.

The Standard File Package recognizes these pseudo-item numbers:

```

CONST {pseudo-items available prior to version 7.0}
sfHookFirstCall      = -1;      {initialize display}
sfHookCharOffset     = $1000;   {offset for character input}
sfHookNullEvent      = 100;     {null event}
sfHookRebuildList    = 101;     {redisplay list}
sfHookFolderPopUp    = 102;     {display parent-directory menu}
sfHookOpenFolder     = 103;     {display contents of }
                                { selected folder or volume}

{additional pseudo-items introduced in version 7.0}

```

Standard File Package

<code>sfHookLastCall</code>	<code>= -2;</code>	<code>{clean up after display}</code>
<code>sfHookOpenAlias</code>	<code>= 104;</code>	<code>{resolve alias}</code>
<code>sfHookGoToDesktop</code>	<code>= 105;</code>	<code>{display contents of desktop}</code>
<code>sfHookGoToAliasTarget</code>	<code>= 106;</code>	<code>{select target of alias}</code>
<code>sfHookGoToParent</code>	<code>= 107;</code>	<code>{display contents of parent}</code>
<code>sfHookGoToNextDrive</code>	<code>= 108;</code>	<code>{display contents of next drive}</code>
<code>sfHookGoToPrevDrive</code>	<code>= 109;</code>	<code>{display contents of previous drive}</code>
<code>sfHookChangeSelection</code>	<code>= 110;</code>	<code>{select target of reply record}</code>
<code>sfHookSetActiveOffset</code>	<code>= 200;</code>	<code>{switch active item}</code>

The Standard File Package uses a set of modal-dialog filter functions (described in “Writing a Modal-Dialog Filter Function” on page 4-203) to map user actions during the dialog onto the defined item numbers. Some of the mapping is indirect. A click of the Open button, for example, is mapped to `sfItemOpenButton` only if a file is selected in the display list. If a folder or volume is selected, the Standard File Package maps the selection onto the pseudo-item `sfHookOpenFolder`.

The lists that follow summarize when various items and pseudo-items are generated and how they are handled. The descriptions indicate the simplest mouse action that generates each item; many of the items can also be generated by keyboard actions, as described in “Keyboard Equivalents” on page 4-182.

Note

Any indicated effects of passing back these constants do not occur until the Standard File Package receives the constant back from your dialog hook function. ♦

Constant descriptions`sfItemOpenButton`

Generated when the user clicks Open or Save while a filename is selected. The Standard File Package fills in the reply record (setting `sfGood` to `TRUE`), removes the dialog box, and returns.

`sfItemCancelButton`

Generated when the user clicks Cancel. The Standard File Package sets `sfGood` to `FALSE`, removes the dialog box, and returns.

`sfItemVolumeUser`

Generated when the user clicks the volume icon or its name. The Standard File Package rebuilds the display list to show the contents of the folder that is one level up the hierarchy (that is, the parent directory of the current parent directory).

`sfItemEjectButton`

Generated when the user clicks Eject. The Standard File Package ejects the volume that is currently selected.

`sfItemDesktopButton`

Generated when the user clicks the Drive button in a customized dialog box defined by one of the earlier procedures. You never receive this item number with the new procedures; when the user clicks the Desktop button, the action is mapped to the item

Standard File Package

`sfHookGoToDesktop`, described later in this section. The Standard File Package displays the contents of the next drive.

`sfItemFileListUser`

Generated when the user clicks an item in the display list. The Standard File Package updates the selection and generates this item for your information.

`sfItemPopUpMenuUser`

Never generated. The Standard File Package's modal-dialog filter function maps clicks on the directory pop-up menu to `sfHookFolderPopUp`, described later in this section.

`sfItemFileNameTextEdit`

Generated when the user clicks the filename field. `TextEdit` and the Standard File Package process mouse clicks in the filename field, but the item number is generated for your information.

`sfItemNewFolderUser`

Generated when the user clicks New Folder. The Standard File Package displays the New Folder dialog box.

The pseudo-items are messages that allow your application and the Standard File Package to communicate and support various features added since the original design of the Standard File Package.

The Standard File Package generates three pseudo-items that give your application the chance to control a customized display.

Constant descriptions

`sfHookFirstCall`

Generated by the Standard File Package as a signal to your dialog hook function that it is about to display a dialog box. If you want to initialize the display, do so when you receive this item. You can specify the current directory either by returning `sfHookGoToDesktop` or by changing the reply record and returning `sfHookChangeSelection`.

`sfHookLastCall`

Generated by the Standard File Package as a signal to your dialog hook function that it is about to remove a dialog box. If you created any structures when the dialog box was first displayed, remove them when you receive this item.

`sfHookNullEvent`

Issued periodically by the Standard File Package if no user action has taken place. Your application can use this null event to perform any updating or periodic processing that might be necessary.

Your application can generate three pseudo-items to request services from the Standard File Package.

Constant descriptions

`sfHookRebuildList`

Returned by your dialog hook function to the Standard File Package when it needs to redisplay the file list. Your application might need to redisplay the list if, for example, it allows the user to change the

Standard File Package

file types to be displayed. The Standard File Package rebuilds and displays the list of files that can be opened.

`sfHookChangeSelection`

Returned by your application to the Standard File Package after your application changes the reply record so that it describes a different file or folder. (You'll need to pass the address of the reply record in the `yourDataPtr` field if you want to do this.) The Standard File Package rebuilds the display list to show the contents of the folder or volume containing the object described in the reply record. It selects the item described in the reply record.

`sfHookSetActiveOffset`

Your application adds this constant to an item number and sends the result to the Standard File Package. The Standard File Package activates that item in the dialog box, making it the target of keyboard input. This constant allows your application to activate a specific field in the dialog box without explicit input from the user.

The Standard File Package's own modal-dialog filter functions generate a number of pseudo-items that allow its dialog hook functions to support various features introduced since the original design of the standard file dialog boxes. Except under extraordinary circumstances, your dialog hook function always passes any of these item numbers back to the Standard File Package for processing.

Constant descriptions`sfHookCharOffset`

The Standard File Package adds this constant to the value of an ASCII character when it's using keyboard input for item selection. The Standard File Package uses the decoded ASCII character to select an entry in the display list.

`sfHookFolderPopUp`

Generated when the user clicks the directory pop-up menu. The Standard File Package displays the pop-up menu showing all parent directories.

`sfHookOpenFolder`

Generated when the user clicks the Open button while a folder or volume is selected in the display list. The Standard File Package rebuilds the display list to show the contents of the folder or volume.

`sfHookOpenAlias`

Generated by the Standard File Package as a signal that the selected item is an alias for another file, folder, or volume. If the selected item is an alias for a file, the Standard File Package resolves the alias, places the file system specification record of the target in the reply record, and returns.

If the selected item is an alias for a folder or volume, the Standard File Package resolves the alias and rebuilds the display list to show the contents of the alias target.

Standard File Package

`sfHookGoToDesktop`

Generated when the user clicks the Desktop button. The Standard File Package displays the contents of the desktop in the display list.

`sfHookGoToAliasTarget`

Generated when the user presses the Option key while opening an item that is an alias. The Standard File Package rebuilds the display list to display the volume or folder containing the alias target and selects the target.

`sfHookGoToParent`

Generated when the user presses Command-Up Arrow (or clicks the volume icon). The Standard File Package rebuilds the display list to show the contents of the folder that is one level up the hierarchy (that is, the parent directory of the current parent directory).

`sfHookGoToNextDrive`

Generated when the user presses Command-Right Arrow. The Standard File Package displays the contents of the next volume.

`sfHookGoToPrevDrive`

Generated when the user presses Command-Left Arrow. The Standard File Package displays the contents of the previous volume.

The `CustomGetFile` and `CustomPutFile` procedures call your dialog hook function for item selections in both the main dialog box and any subsidiary dialog boxes (such as the dialog box for naming a new folder while saving a document through `CustomPutFile`). To determine whether the dialog record describes the main dialog box or a subsidiary dialog box, check the value of the `refCon` field in the window record in the dialog record.

Note

Prior to system software version 7.0, the Standard File Package did not call your dialog hook function during subsidiary dialog boxes. Dialog hook functions for the new `CustomGetFile` and `CustomPutFile` procedures must check the dialog window's `refCon` field to determine the target of the dialog record. ♦

The defined values for the `refCon` field represent the Standard File dialog boxes.

CONST

```
sfMainDialogRefCon      = 'stdf';  {main dialog box}
sfNewFolderDialogRefCon = 'nfdr';  {New Folder dialog box}
sfReplaceDialogRefCon   = 'rplc';  {name conflict dialog box}
sfStatWarnDialogRefCon  = 'stat';  {stationery warning}
sfErrorDialogRefCon     = 'err ';  {general error report}
sfLockWarnDialogRefCon  = 'lock';  {software lock warning}
```

Constant descriptions

<code>sfMainDialogRefCon</code>	The main dialog box, either Open or Save.
<code>sfNewFolderDialogRefCon</code>	The New Folder dialog box.

Standard File Package

<code>sfReplaceDialogRefCon</code>	The dialog box requesting verification for replacing a file of the same name.
<code>sfStatWarnDialogRefCon</code>	The dialog box warning that the user is opening the master copy of a stationery pad, not a piece of stationery.
<code>sfErrorDialogRefCon</code>	A dialog box reporting a general error.
<code>sfLockWarnDialogRefCon</code>	The dialog box warning that the user is opening a locked file and won't be allowed to save any changes.

Listing 4-9 defines a dialog hook function that handles user selections in the customized Open dialog box illustrated in Figure 4-7. Note that this dialog hook function handles selections only in the main dialog box, not in any subsidiary dialog boxes.

Listing 4-9 A sample dialog hook function

```

FUNCTION MyDlgHook (item: Integer; theDialog: DialogPtr; myDataPtr: Ptr):
    Integer;
VAR
    myType:      Integer;      {menu item selected}
    myHandle:    Handle;       {needed for GetDItem}
    myRect:      Rect;         {needed for GetDItem}
    myIgnore:    Integer;      {needed for GetDItem; ignored}
CONST
    kMyPopUpItem = 10;         {item number of File Type pop-up menu}
BEGIN
    MyDlgHook := item;         {by default, return the item passed in}
    IF GetWRefCon(WindowPtr(theDialog)) <> LongInt(sfMainDialogRefCon) THEN
        Exit(MyDlgHook);       {this function is only for main dialog}

    {Do processing of pseudo-items and your own additional item.}
    CASE item OF
        sfHookFirstCall:      {pseudo-item: first time function called}
            BEGIN
                GetDItem(theDialog, kPopUpItem, myType, myHandle, myRect);
                SetCtlValue(ControlHandle(myHandle), gCurrentType);
                MyDlgHook := sfHookNullEvent;
            END;
        kMyPopUpItem:         {user selected File Type pop-up menu}
            BEGIN
                GetDItem(theDialog, item, myIgnore, myHandle, myRect);
                myType := GetCtlValue(ControlHandle(myHandle));
                IF myType <> gCurrentType THEN
                    BEGIN
                        gCurrentType := myType;
                        MyDlgHook := sfHookRebuildList;
                    END;
            END;
    END;

```

Standard File Package

```

END;
OTHERWISE
    ;                                {ignore all other items}
END;
END;

```

The pop-up menu is stored as a control in the application's resource fork. Values stored in the resource determine the appearance of the control, such as the pop-up title text and the menu associated with the control. The Dialog Manager's `ModalDialog` procedure takes care of drawing the box around the pop-up menu and the title of the dialog box. When the dialog hook function is first called, it simply retrieves a handle to that control and sets the value of the pop-up control to the current menu item (stored in the global variable `gCurrentType`). The `MyDlgHook` function then returns `sfHookNullEvent` to indicate that no further processing is required.

When the user clicks the pop-up menu control, `ModalDialog` calls the standard control definition function associated with it. If the user makes a selection in the pop-up menu, `MyDlgHook` is called with the `item` parameter equal to `kPopUpItem`. Your dialog hook function needs simply to determine the current value of the control and respond accordingly. In this case, if the user has selected a new file type, the global variable `gCurrentType` is updated to reflect the new selection, and `MyDlgHook` returns `sfHookRebuildList` to cause the Standard File Package to rebuild the list of files and folders displayed in the dialog box.

For complete details on handling pop-up menus, see the chapters "Control Manager" and "Menu Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

Writing a Modal-Dialog Filter Function

A **modal-dialog filter function** controls events closer to their source by filtering the events received from the Event Manager. The Standard File Package itself contains an internal modal-dialog filter function that maps keypresses and other user input onto the equivalent dialog box items. If you also want to process events at this level, you can supply your own filter function.

Note

You can supply a modal-dialog filter function only when you use one of the procedures that displays a customized dialog box (that is, `CustomGetFile`, `CustomPutFile`, `SFPGetFile`, or `SFPPutFile`). ♦

Your modal-dialog filter function determines how the Dialog Manager procedure `ModalDialog` filters events. The `ModalDialog` procedure retrieves events by calling the Event Manager function `GetNextEvent`. As just indicated, the Standard File Package contains an internal filter function that performs some preliminary processing on each event it receives. If you provide a modal-dialog filter function, `ModalDialog` calls your filter function after it calls the internal Standard File Package filter function and before it sends the event to your dialog hook function.

You might provide a modal-dialog filter function for several reasons. If you have customized the Open or Save dialog boxes by adding one or more items, you might want

Standard File Package

to map some of the user's keypresses to those items in the same way that the internal filter function maps certain keypresses to existing items.

Another reason to provide a modal-dialog filter function is to avoid a problem that can arise if an update event is received for one of your application's windows while a Standard File Package dialog box is displayed.

Note

The problem described in the following paragraph occurs only in system software versions earlier than version 7.0. The internal modal-dialog filter function installed by the Standard File Package when running in version 7.0 and later avoids the problem by passing the update event to your dialog filter and, if your filter doesn't handle the event, mapping it to a null event. ♦

When `ModalDialog` calls `GetNextEvent` and receives the update event, `ModalDialog` does not know how to respond to it and therefore passes the update event to the Standard File Package's internal filter function. The internal filter function cannot handle the update event either. As a result, if you do not provide your own modal-dialog filter function that handles the update event, that event is never cleared. The next time `ModalDialog` calls `GetNextEvent`, it receives the same update event. `ModalDialog` never receives a null event, so your dialog hook function never performs any processing in response to the `sfHookNullEvent` pseudo-item. You can solve this problem by providing a modal-dialog filter function that handles the update event or changes it to a null event. See Listing 4-10 for details.

A modal-dialog filter function used with `SFPGetFile` and `SFPPutFile` is declared like any filter function passed to `ModalDialog`. Your function is passed a pointer to the dialog record, a pointer to the event record, and the item number. (The modal-dialog filter function is described in the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.)

```
FUNCTION MyModalFilter (theDialog: DialogPtr;
                       VAR theEvent: EventRecord;
                       VAR itemHit: Integer): Boolean;
```

The modal-dialog filter function used with `CustomGetFile` and `CustomPutFile` requires an additional parameter, a pointer (`myDataPtr`) to the data received from your application, if any.

```
FUNCTION MyModalFilterYD (theDialog: DialogPtr;
                         VAR theEvent: EventRecord;
                         VAR itemHit: Integer;
                         myDataPtr: Ptr): Boolean;
```

Your modal-dialog filter function returns a Boolean value that reports whether it handled the event. If your function returns a value of `FALSE`, `ModalDialog` processes the event through its own filters. If your function returns a value of `TRUE`, `ModalDialog` returns with no further action.

Standard File Package

The `CustomGetFile` and `CustomPutFile` procedures call your filter function to process events in both the main dialog box and any subsidiary dialog boxes (such as the dialog box for naming a new folder while saving a document through `CustomPutFile`). To determine whether the dialog record describes the main dialog box or a subsidiary dialog box, check the value of the `refCon` field in the window record in the dialog record, as described in “Writing a Dialog Hook Function” beginning on page 4-196.

Listing 4-10 shows how to define a modal-dialog filter function that prevents update events from clogging the event queue.

Listing 4-10 A sample modal-dialog filter function

```
FUNCTION MyModalFilter (theDialog: DialogPtr; VAR theEvent: EventRecord;
                      VAR itemHit: Integer): Boolean;

BEGIN
    MyModalFilter := FALSE;           {we haven't handled the event yet}
    IF theEvent.what = updateEvt THEN
        IF IsAppWindow(WindowPtr(theEvent.message)) THEN
            BEGIN
                DoUpdateEvent(WindowPtr(theEvent.message));
                MyModalFilter := TRUE;   {we have handled the event}
            END;
        END;
    END;
```

If this filter function receives an update event for a window other than the Standard File Package dialog box, it calls the application's routine for handling update events (`DoUpdateEvent`) and returns `TRUE` to indicate that the event has been handled. See the chapters “Event Manager” and “Window Manager” in *Inside Macintosh: Macintosh Toolbox Essentials* for complete details on handling update events.

Writing an Activation Procedure

The **activation procedure** controls the highlighting of dialog items that are defined by your application and can receive keyboard input. Ordinarily, you need to supply an activation procedure only if your application builds a list from which the user can select entries. The Standard File Package supplies the activation procedure for the file display list and for all `TextEdit` fields. You can also use the activation procedure to keep track of which field is receiving keyboard input, if your application needs that information.

The target of keyboard input is called the **active field**. The two standard keyboard-input fields are the filename field (present only in Save dialog boxes) and the display list. Unless you override it through your own dialog hook function, the Standard File Package handles the highlighting of its own items and `TextEdit` fields. When the user changes the keyboard target by pressing the mouse button or the Tab key, the Standard File Package

Standard File Package

calls your activation procedure twice: the first call specifies which field is being deactivated, and the second specifies which field is being activated. Your application is responsible for removing the highlighting when one of its fields becomes inactive and for adding the highlighting when one of its fields becomes active. The Standard File Package can handle the highlighting of all TextEdit fields, even those defined by your application.

The activation procedure receives four parameters: a dialog pointer, a dialog item number, a Boolean value that specifies whether the field is being activated (TRUE) or deactivated (FALSE), and a pointer to your own data.

```
PROCEDURE MyActivateProc (theDialog: DialogPtr; itemNo: Integer;
                          activating: Boolean; myDataPtr: Ptr);
```

Setting the Current Directory

The first time your application calls one of the Standard File Package routines, the default current directory (that is, the directory whose contents are listed in the dialog box) is determined by the way in which your application was launched.

- If the user launched your application directly (perhaps by double-clicking its icon in the Finder), the default directory is the directory in which your application is located.
- If the user launched your application indirectly (perhaps by double-clicking one of your application's document icons), the default directory is the directory in which that document is located.

At each subsequent call to one of the Standard File Package routines, the default current directory is simply the directory that was current when the user completed the previous dialog box. You can use the function `GetSFCurDir` defined in Listing 4-11 to determine the current directory.

Listing 4-11 Determining the current directory

```
FUNCTION GetSFCurDir: LongInt;
TYPE
    LongIntPtr = ^LongInt;
CONST
    CurDirStore = $398;
BEGIN
    GetSFCurDir := LongIntPtr(CurDirStore)^;
END;
```

Standard File Package

You can use the `GetSFCurVol` function defined in Listing 4-12 to determine the current volume.

Listing 4-12 Determining the current volume

```
FUNCTION GetSFCurVol: Integer;
TYPE
  IntPtr = ^Integer;
CONST
  SFSaveDisk = $214;
BEGIN
  GetSFCurVol := -IntPtr(SFSaveDisk)^;
END;
```

If necessary, you can change the default current directory and volume. For example, when the user needs to select a dictionary file for a spell-checking application, the application might set the current directory to a directory containing document-specific dictionary files. This saves the user from having to navigate the directory hierarchy from the directory containing documents to that containing dictionary files. You can use the procedure `SetSFCurDir` defined in Listing 4-13 to set the current directory.

Listing 4-13 Setting the current directory

```
PROCEDURE SetSFCurDir (dirID: LongInt);
TYPE
  LongIntPtr = ^LongInt;
CONST
  CurDirStore = $398;
BEGIN
  LongIntPtr(CurDirStore)^ := dirID;
END;
```

You can use the procedure `SetSFCurVol` defined in Listing 4-14 to set the current volume.

Listing 4-14 Setting the current volume

```
PROCEDURE SetSFCurVol (vRefNum: Integer);
TYPE
  IntPtr = ^Integer;
CONST
  SFSaveDisk = $214;
```

Standard File Package

```
BEGIN
    IntPtr(SFSaveDisk)^ := -vRefNum;
END;
```

Note

Most applications don't need to alter the default current directory or volume. ♦

If you are using the enhanced Standard File Package routines, you can set the current directory by filling in the fields of the file system specification in the reply record passed to CustomGetFile or CustomPutFile. You do this within your dialog hook function. Listing 4-15 defines a dialog hook function that makes the currently active System Folder the current directory.

Listing 4-15 Setting the current directory

```
FUNCTION MyDlgHook (item: Integer; theDialog: DialogPtr; myDataPtr: Ptr):
    Integer;

VAR
    myReplyPtr:   StandardFileReplyPtr;
    foundVRefNum: Integer;
    foundDirID:   LongInt;
    myErr:        OSErr;
BEGIN
    MyDlgHook := item;           {by default, return the item passed in}
    IF GetWRefCon(WindowPtr(theDialog)) <> LongInt(sfMainDialogRefCon) THEN
        Exit(MyDlgHook);        {this function is only for main dialog box}

    CASE item OF
        sfHookFirstCall:        {pseudo-item: first time function called}
            BEGIN
                myReplyPtr := StandardFileReplyPtr(myDataPtr);
                myErr := FindFolder(kOnSystemDisk, kSystemFolderType,
                                    kDontCreateFolder, foundVRefNum, foundDirID);
                IF myErr = noErr THEN
                    BEGIN
                        myReplyPtr^.sfFile.parID := foundDirID;
                        myReplyPtr^.sfFile.vRefNum := foundVRefNum;
                        MyDlgHook := sfHookChangeSelection;
                    END;
                END;
            END;
        OTHERWISE
```



```

;                                     {ignore all other items}
END;
END;

```

This dialog hook function installs the System Folder's volume reference number and parent directory ID into the file system specification whose address is passed in the `myDataPtr` parameter. Because the dialog hook function returns the constant `sfHookChangeSelection` the first time it is called (that is, in response to the `sfHookFirstCall` pseudo-item), the Standard File Package sets the current directory to the indicated directory when the dialog box is displayed.

Selecting a Directory

You can present the recommended user interface for selecting a directory by calling the `CustomGetFile` procedure and passing it the addresses of a custom file filter function and a dialog hook function. See “Selecting Volumes and Directories” on page 4-184 for a description of the appearance and behavior of the directory selection dialog box.

The file filter function used to select directories is quite simple; it ensures that only directories, not files, are listed in the dialog box displayed by `CustomGetFile`. Listing 4-16 defines a file filter function you can use for this purpose.

Listing 4-16 A file filter function that lists only directories

```

FUNCTION MyCustomFileFilter (pb: CInfoPBPtr; myDataPtr: Ptr): Boolean;
CONST
    kFolderBit = 4;                                     {bit set in ioFlAttrib for a directory}
BEGIN                                                  {list directories only}
    MyCustomFileFilter := NOT BTst(pb^.ioFlAttrib, kFolderBit);
END;

```

The function `MyCustomFileFilter` simply inspects the appropriate bit in the file attributes (`ioFlAttrib`) field of the catalog information parameter block passed to it. If the directory bit is set, the file filter function returns `FALSE`, indicating that the item should appear in the list; otherwise, the file filter function returns `TRUE` to exclude the item from the list. Because a volume is identified via its root directory, volumes also appear in the list of items in the dialog box.

The title of the Select button should identify which directory is available for selection. You can use the `SetButtonTitle` procedure defined in Listing 4-17 to set the title of a button.

Your dialog hook function calls the `SetButtonTitle` procedure to copy the truncated title of the selected item into the Select button. This title eliminates possible user confusion about which directory is available for selection. If no item in the list is selected, the dialog hook function uses the name of the directory shown in the pop-up menu as the title of the Select button.

Listing 4-17 Setting a button's title

```

PROCEDURE SetButtonTitle (ButtonHdl: Handle; name: Str255; ButtonRect: Rect);
VAR
    result: Integer;    {result of TruncString}
    width: Integer;     {width available for name of directory}
BEGIN
    gPrevSelectedName := name;
    WITH ButtonRect DO
        width := (right - left) - (StringWidth('Select "') + CharWidth(' '));
        result := TruncString(width, name, smTruncMiddle);
        SetCTitle(ControlHandle(ButtonHdl), CONCAT('Select ', name, ' '));
        ValidRect(ButtonRect);
    END;
END;

```

The `SetButtonTitle` procedure is passed a handle to the button whose title is to be changed, the name of the directory available for selection, and the button's enclosing rectangle. The global variable `gPrevSelectedName` holds the full directory name, before truncation.

A dialog hook function manages most of the process of letting the user select a director. Listing 4-18 defines a dialog hook function that handles user selections in the dialog box.

Listing 4-18 Handling user selections in the directory selection dialog box

```

FUNCTION MyDlgHook (item: Integer; theDialog: DialogPtr; myDataPtr: Ptr):
    Integer;
CONST
    kGetDirBTN = 10;          {Select directory button}
TYPE
    SFRPtr = ^StandardFileReply;
VAR
    myType: Integer;          {menu item selected}
    myHandle: Handle;          {needed for GetDItem}
    myRect: Rect;              {needed for GetDItem}
    myName: Str255;
    myPB: CInfoPBRec;
    mySFRPtr: SFRPtr;
    myErr: OSErr;
BEGIN
    MyDlgHook := item;        {default, except in special cases below}
    IF GetWRefCon(WindowPtr(theDialog)) <> LongInt(sfMainDialogRefCon) THEN
        Exit(MyDlgHook);      {this function is only for main dialog box}

    GetDItem(theDialog, kGetDirBTN, myType, myHandle, myRect);
    IF item = sfHookFirstCall THEN

```

Standard File Package

```

BEGIN
    {Determine current folder name and set title of Select button.}
    WITH myPB DO
        BEGIN
            ioCompletion := NIL;
            ioNamePtr := @myName;
            ioVRefNum := GetSFCurVol;
            ioFDirIndex := - 1;
            ioDirID := GetSFCurDir;
        END;
        myErr := PBGetCatInfo(@myPB, FALSE);
        SetButtonTitle(myHandle, myName, myRect);
    END
ELSE
    BEGIN
        {Get mySFRPtr from 3rd parameter to hook function.}
        mySFRPtr := SFRPtr(myDataPtr);
        {Track name of folder that can be selected.}
        IF (mySFRPtr^.sfIsFolder) OR (mySFRPtr^.sfIsVolume) THEN
            myName := mySFRPtr^.sfFile.name
        ELSE
            BEGIN
                WITH myPB DO
                    BEGIN
                        ioCompletion := NIL;
                        ioNamePtr := @myName;
                        ioVRefNum := mySFRPtr^.sfFile.vRefNum;
                        ioFDirIndex := -1;
                        ioDrDirID := mySFRPtr^.sfFile.parID;
                    END;
                    myErr := PBGetCatInfo(@myPB, FALSE);
                END;
                {Change directory name in button title as needed.}
                IF myName <> gPrevSelectedName THEN
                    SetButtonTitle(myHandle, myName, myRect);

                CASE item OF
                    kGetDirBTN:                {force return by faking a cancel}
                        MyDlgHook := sfItemCancelButton;
                    sfItemCancelButton:
                        gDirSelectionFlag := FALSE; {flag no directory was selected}
                OTHERWISE
                    ;
                END; {CASE}
            END;
        END;
    END;
END;

```

Standard File Package

The `MyDlgHook` dialog hook function defined in Listing 4-18 calls the File Manager function `PBGetCatInfo` to retrieve the name of the directory to be selected. When the dialog hook function is first called (that is, when `item` is set to `sfHookFirstCall`), `MyDlgHook` determines the current volume and directory by calling the functions `GetSFCurVol` and `GetSFCurDir`. When `MyDlgHook` is called each subsequent time, `MyDlgHook` calls `PBGetCatInfo` with the volume reference number and directory ID of the previously opened directory.

When the user clicks the Select button, `MyDlgHook` returns the item `sfItemCancelButton`. When the user clicks the real Cancel button, `MyDlgHook` sets the global variable `gDirSelectionFlag` to `FALSE`, indicating that the user didn't select a directory. The function `DoGetDirectory` uses that variable to distinguish between clicks of Cancel and clicks of Select.

The function `DoGetDirectory` defined in Listing 4-19 uses the file filter function and the dialog hook functions defined above to manage the directory selection dialog box. On exit, `DoGetDirectory` returns a standard file reply record describing the selected directory.

Listing 4-19 Presenting the directory selection dialog box

```
FUNCTION DoGetDirectory: StandardFileReply;
VAR
    myReply:           StandardFileReply;
    myTypes:           SFTypelist;           {types of files to display}
    myPoint:           Point;                {upper-left corner of box}
    myNumTypes:        Integer;
    myModalFilter:     ModalFilterYDProcPtr;
    myActiveList:      Ptr;
    myActivateProc:    ActivateYDProcPtr;
    myName:            Str255;
CONST
    rGetDirectoryDLOG = 128;                 {resource ID of custom dialog box}
BEGIN
    gPrevSelectedName := '';                 {initialize name of previous selection}
    gDirSelectionFlag := TRUE;               {initialize directory selection flag}
    myNumTypes := -1;                        {pass all types of files to file filter}
    myPoint.h := -1;                         {center dialog box on screen}
    myPoint.v := -1;
    myModalFilter := NIL;
    myActiveList := NIL;
    myActivateProc := NIL;

    CustomGetFile(@MyCustomFileFilter, myNumTypes, myTypes, myReply,
                  rGetDirectoryDLOG, myPoint, @MyDlgHook, myModalFilter,
                  myActiveList, myActivateProc, @myReply);
```

Standard File Package

```

{Get the name of the directory.}
IF gDirSelectionFlag AND myReply.sfIsVolume THEN
    myName := Concat(myReply.sfFile.name, ':')
ELSE
    myName := myReply.sfFile.name;

IF gDirSelectionFlag AND myReply.sfIsVolume THEN
    myReply.sfFile.name := myName
ELSE IF gDirSelectionFlag THEN
    myReply.sfFile.name := gPrevSelectedName;
gDirSelectionFlag := FALSE;
DoGetDirectory := myReply;
END;

```

The `DoGetDirectory` function initializes the two global variables `gPrevSelectedName` and `gDirSelectionFlag`. As you have seen, these two variables are used by the custom dialog hook function. Then `DoGetDirectory` calls `CustomGetFile` to display the directory selection dialog box and handle user selections. When the user selects a directory or clicks the Cancel button, the dialog hook function returns `sfItemCancelButton` and `CustomGetFile` exits. At that point, the reply record contains information about the last item selected in the list of available items.

Selecting a Volume

You can present the recommended user interface for selecting a volume by calling the `CustomGetFile` procedure and passing it the addresses of a custom file filter function and a dialog hook function. See “Selecting Volumes and Directories” on page 4-184 for a description of the appearance and behavior of the volume selection dialog box.

The file filter function used to select volumes is quite simple; it ensures that only volumes, not files or directories, are listed in the dialog box displayed by `CustomGetFile`. Listing 4-16 defines a file filter function you can use to do this.

Listing 4-20 A file filter function that lists only volumes

```

FUNCTION MyCustomFileFilter (pb: CInfoBPBPtr; myDataPtr: Ptr): Boolean;
CONST
    kFolderBit = 4;                {bit set in ioFlAttrib for a directory}
BEGIN                             {list volumes only}
    MyCustomFileFilter := TRUE;    {assume you don't list the item}
    IF BTst(pb^.ioFlAttrib, kFolderBit) AND (pb^.ioDrParID = fsRtParID) THEN
        MyCustomFileFilter := FALSE;
    END;

```

Standard File Package

The function `MyCustomFileFilter` inspects the appropriate bit in the file attributes (`ioFlAttrib`) field of the catalog information parameter block passed to it. If the directory bit is set, `MyCustomFileFilter` checks whether the parent directory ID of the directory is equal to `fsRtParID`, which is always the parent directory ID of a volume's root directory. If it is, the file filter function returns `FALSE`, indicating that the item should appear in the list of volumes; otherwise, the file filter function returns `TRUE` to exclude the item from the list.

A dialog hook function for handling the items in the volume selection dialog box is defined in Listing 4-21.

Listing 4-21 Handling user selections in the volume selection dialog box

```
FUNCTION MyDlgHook (item: Integer; theDialog: DialogPtr; myDataPtr: Ptr):
    Integer;

VAR
    myType:      Integer;      {menu item selected}
    myHandle:    Handle;       {needed for GetDItem}
    myRect:      Rect;         {needed for GetDItem}
    myName:      Str255;       {new title for Open button}
BEGIN
    MyDlgHook := item;        {default, except in special cases below}
    IF GetWRefCon(WindowPtr(theDialog)) <> LongInt(sfMainDialogRefCon) THEN
        Exit(MyDlgHook);      {this function is only for main dialog box}

    CASE item OF
        sfHookFirstCall:
            BEGIN
                {Set button title and go to desktop.}
                myName := 'Select';
                GetDItem(theDialog, sfItemOpenButton, myType, myHandle, myRect);
                SetCTitle(ControlHandle(myHandle), myName);
                MyDlgHook := sfHookGoToDesktop;
            END;
        sfHookGoToDesktop:      {map Cmd-D to a null event}
            MyDlgHook := sfHookNullEvent;
        sfHookChangeSelection:
            MyDlgHook := sfHookGoToDesktop;
        sfHookGoToNextDrive:    {map Cmd-Left Arrow to a null event}
            MyDlgHook := sfHookNullEvent;
        sfHookGoToPrevDrive:    {map Cmd-Right Arrow to a null event}
            MyDlgHook := sfHookNullEvent;
        sfItemOpenButton, sfHookOpenFolder:
            MyDlgHook := sfItemOpenButton;
    OTHERWISE
```

```

;
END;
END;

```

You can prompt the user to select a volume by calling the function `DoGetVolume` defined in Listing 4-22.

Listing 4-22 Presenting the volume selection dialog box

```

FUNCTION DoGetVolume: StandardFileReply;
VAR
    myReply:           StandardFileReply;
    myTypes:           SFTYPEList;           {types of files to display}
    myPoint:           Point;                {upper-left corner of box}
    myNumTypes:        Integer;
    myModalFilter:      ModalFilterYDProcPtr;
    myActiveList:       Ptr;
    myActivateProc:     ActivateYDProcPtr;
CONST
    rGetVolumeDLOG     = 129;                {resource ID of custom dialog box}
BEGIN
    myNumTypes := -1;                        {pass all types of files}
    myPoint.h := -1;                         {center dialog box on screen}
    myPoint.v := -1;
    myModalFilter := NIL;
    myActiveList := NIL;
    myActivateProc := NIL;

    CustomGetFile(@MyCustomFileFilter, myNumTypes, myTypes, myReply,
                  rGetVolumeDLOG, myPoint, @MyDlgHook, myModalFilter,
                  myActiveList, myActivateProc, @myReply);

    DoGetVolume := myReply;
END;

```

Using the Original Procedures

The Standard File Package still recognizes all procedures available before system software version 7.0 (`SFGetFile`, `SFPutFile`, `SFPGetFile`, and `SFPPutFile`). It displays the new interface for all applications that don't customize the dialog boxes in incompatible ways (that is, applications that specify both the dialog hook and the modal-dialog filter pointers as `NIL` and that specify no alternative dialog ID).

Standard File Package

When the Standard File Package can't use the enhanced dialog box layout because an application customized the dialog box with the earlier procedures, it nevertheless makes some changes to the display:

- It changes the label of the Drive button to Desktop and makes the desktop the root of the display.
- It moves the volume icon slightly to the right, to make room for selection highlighting around the display list field.

If, however, a customized dialog box has suppressed the file display list (by specifying coordinates outside of the dialog box), the Standard File Package uses the earlier interface, on the assumption that the dialog box is designed for volume selection.

If you need to use the procedures available before system software version 7.0, you need to be aware of a number of differences between those procedures and the enhanced procedures. These are the most important differences:

- The original procedures do not recognize some pseudo-items under previous system software versions. For example, the pseudo-item `sfHookLastCall` is not used before version 7.0. See the comments under “Constants” in “” (beginning on page 4-234) for information on which pseudo-items are universally available.
- The original standard file reply record (type `SFReply`) returns a working directory reference number, not a volume reference number. Typically, you should immediately convert that number to a volume reference number and directory ID using `GetWDInfo` or `PBGetWDInfo`. Then close the working directory by calling `CloseWD` or `PBCloseWD`. For details on these functions, see the chapter “File Manager” in this book.
- Dialog hook functions used with the original procedures are not passed a `myDataPtr` parameter.

Standard File Package Reference

This section describes the data structures and routines that are specific to the Standard File Package. The “Data Structures” section shows the Pascal data structures for the original and the enhanced Standard File reply records. The section “Standard File Package Routines” describes routines for opening and saving files. The section “Application-Defined Routines” describes the routines that your application can define to customize the operations of the Standard File Package routines.

Data Structures

The Standard File Package exchanges information with your application using a standard file reply record. If you use the procedures introduced in system software version 7.0, you use a reply record of type `StandardFileReply`. If you use the procedures available before version 7.0, you must use a reply record of type `SFReply`.

Enhanced Standard File Reply Record

When you use one of the procedures `StandardPutFile`, `StandardGetFile`, `CustomPutFile`, or `CustomGetFile`, you pass a reply record of type `StandardFileReply`.

```
TYPE StandardFileReply =
RECORD
    sfGood:          Boolean;      {TRUE if user did not cancel}
    sfReplacing:     Boolean;      {TRUE if replacing file with same name}
    sfType:          OSType;       {file type}
    sfFile:          FSSpec;       {selected file, folder, or volume}
    sfScript:        ScriptCode;   {script of file, folder, or volume name}
    sfFlags:         Integer;      {Finder flags of selected item}
    sfIsFolder:      Boolean;      {selected item is a folder}
    sfIsVolume:      Boolean;      {selected item is a volume}
    sfReserved1:     LongInt;      {reserved}
    sfReserved2:     Integer;      {reserved}
END;
```

Field descriptions

<code>sfGood</code>	Reports whether the reply record is valid. The value is <code>TRUE</code> after the user clicks <code>Save</code> or <code>Open</code> ; <code>FALSE</code> after the user clicks <code>Cancel</code> . When the user has completed the dialog box, the other fields in the reply record are valid only if the <code>sfGood</code> field contains <code>TRUE</code> .
<code>sfReplacing</code>	Reports whether a file to be saved replaces an existing file of the same name. This field is valid only after a call to the <code>StandardPutFile</code> or <code>CustomPutFile</code> procedure. When the user assigns a name that duplicates that of an existing file, the Standard File Package asks for verification by displaying a subsidiary dialog box (illustrated in Figure 4-4, page 4-181). If the user verifies the name, the Standard File Package sets the <code>sfReplacing</code> field to <code>TRUE</code> and returns to your application; if the user cancels the overwriting of the file, the Standard File Package returns to the main dialog box. If the name does not conflict with an existing name, the Standard File Package sets the field to <code>FALSE</code> and returns.
<code>sfType</code>	Contains the file type of the selected file. (File types are described in the chapter “Finder Interface” in <i>Inside Macintosh: Macintosh Toolbox Essentials</i> .) Only <code>StandardGetFile</code> and <code>CustomGetFile</code> return a file type in this field.
<code>sfFile</code>	Describes the selected file, folder, or volume with a file system specification record, which contains a volume reference number, parent directory ID, and name. (See the chapter “File Manager” in this book for a complete description of the file system specification record.) If the selected item is an alias for another item, the Standard

Standard File Package

	File Package resolves the alias and places the file system specification record for the target in the <code>sfFile</code> field when the user completes the dialog box. If the selected file is a stationery pad, the reply record describes the file itself, not a copy of the file.
<code>sfScript</code>	Identifies the script in which the name of the document is to be displayed. (This information is used by the Finder and by the Standard File Package.) A script code of <code>smSystemScript</code> (-1) represents the default system script.
<code>sfFlags</code>	Contains the Finder flags from the Finder information record in the catalog entry for the selected file. (See the chapter “Finder Interface” in <i>Inside Macintosh: Macintosh Toolbox Essentials</i> for a description of the Finder flags.) This field is returned only by <code>StandardGetFile</code> and <code>CustomGetFile</code> . If your application supports stationery, it should check the stationery bit in the Finder flags to determine whether to treat the selected file as stationery. Unlike the Finder, the Standard File Package does not automatically create a document from a stationery pad and pass your application the new document. If the user opens a stationery document from within an application that does not support stationery, the Standard File Package displays a dialog box warning the user that the master copy is being opened.
<code>sfIsFolder</code>	Reports whether the selected item is a folder (TRUE) or a file or volume (FALSE). This field is meaningful only during the execution of a dialog hook function.
<code>sfIsVolume</code>	Reports whether the selected item is a volume (TRUE) or a file or folder (FALSE). This field is meaningful only during the execution of a dialog hook function.
<code>sfReserved1</code>	Reserved.
<code>sfReserved2</code>	Reserved.

Original Standard File Reply Record

When you use one of the original Standard File Package procedures `SFPutFile`, `SFGetFile`, `SFPPutFile`, or `SFPGetFile`, you pass a reply record of type `SFReply`.

```

SFReply =
RECORD
    good:      Boolean;      {TRUE if user did not cancel}
    copy:      Boolean;      {reserved}
    fType:     OSType;       {file type}
    vRefNum:   Integer;      {working directory reference number}
    version:   Integer;      {reserved}
    fName:     Str63;        {filename}
END;
```

Standard File Package

Field descriptions

<code>good</code>	Reports whether the reply record is valid. The value is <code>TRUE</code> after the user clicks <code>Save</code> or <code>Open</code> ; <code>FALSE</code> after the user clicks <code>Cancel</code> . When the user has completed the dialog box, the other fields in the reply record are valid only if the value of <code>good</code> is <code>TRUE</code> .
<code>copy</code>	Reserved.
<code>fType</code>	Contains the file type of the selected file. (File types are described in the chapter “Finder Interface” of <i>Inside Macintosh: Macintosh Toolbox Essentials</i> .) Only <code>SFGetFile</code> and <code>SFPGetFile</code> return a file type in this field.
<code>vRefNum</code>	Contains the working directory reference number of the selected file.
<code>version</code>	Reserved.
<code>fName</code>	Contains the name of the selected file.

Note

In spite of its name, the `vRefNum` field does not contain a volume reference number. Instead, it contains a working directory reference number, which encodes both the volume reference number and the parent directory ID of the selected file. You can obtain the volume reference number and directory ID of the file by calling `GetWDInfo` or `PBGetWDInfo`. See the chapter “File Manager” in this book for details about working directory reference numbers. ♦

Standard File Package Routines

This section describes the routines you can use to prompt the user for a file's name and location after a request to save or open a file. If your application is designed to run in system software versions prior to version 7.0, you must use either `SFGetFile` or `SFPGetFile` when opening a file and either `SFPutFile` or `SFPPutFile` when saving a file.

If your application is designed to take advantage of features introduced in system software version 7.0 or later, you can use the new routines intended to simplify the code required to elicit a filename from the user. The `StandardPutFile` and `StandardGetFile` procedures are simplified versions of the original procedures for handling the user interface during the storing and retrieving of files. The `CustomPutFile` and `CustomGetFile` procedures are customizable versions of the same procedures.

Saving Files

You can use the `StandardPutFile` procedure to present the standard user interface when the user asks to save a file. If you need to add elements to the default dialog boxes or exercise greater control over user actions in the dialog box, use `CustomPutFile`.

Standard File Package

If your application is designed to execute in system software versions earlier than version 7.0, you can use the corresponding procedures `SFPutFile` and `SFPPutFile`.

StandardPutFile

You can use the `StandardPutFile` procedure to display the default Save dialog box when the user is saving a file.

```
PROCEDURE StandardPutFile (prompt: Str255; defaultName: Str255;
                          VAR reply: StandardFileReply);
```

<code>prompt</code>	The prompt message to be displayed over the text field.
<code>defaultName</code>	The initial name of the file.
<code>reply</code>	The reply record, which <code>StandardPutFile</code> fills in before returning.

DESCRIPTION

The `StandardPutFile` procedure presents a dialog box through which the user specifies the name and location of a file to be written to. The dialog box is centered on the screen. While the dialog box is active, `StandardPutFile` gets and handles events until the user completes the interaction, either by selecting a name and authorizing the save or by canceling the save. The `StandardPutFile` procedure returns the user's input in a record of type `StandardFileReply`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `StandardPutFile` are

Trap macro	Selector
<code>_Pack3</code>	<code>\$0005</code>

SPECIAL CONSIDERATIONS

The `StandardPutFile` procedure is not available in all versions of system software. Use the `Gestalt` function to determine whether `StandardPutFile` is available before calling it.

Because `StandardPutFile` may move memory, you should not call it at interrupt time.

CustomPutFile

Use the `CustomPutFile` procedure when your application requires more control over the Save dialog box than is possible using `StandardPutFile`.

```
PROCEDURE CustomPutFile (prompt: Str255; defaultName: Str255;
    VAR reply: StandardFileReply;
    dlgID: Integer; where: Point;
    dlgHook: DlgHookYDProcPtr;
    filterProc: ModalFilterYDProcPtr;
    activeList: Ptr;
    activateProc: ActivateYDProcPtr;
    yourDataPtr: UNIV Ptr);
```

<code>prompt</code>	The prompt message to be displayed over the text field.
<code>defaultName</code>	The initial name of the file.
<code>reply</code>	The reply record, which <code>CustomPutFile</code> fills in before returning.
<code>dlgID</code>	The resource ID of a customized dialog template. To use the standard template, set this parameter to 0.
<code>where</code>	The upper-left corner of the dialog box, in global coordinates. If you specify the point (-1,-1), <code>CustomPutFile</code> automatically centers the dialog box on the screen.
<code>dlgHook</code>	A pointer to your dialog hook function, which handles item selections received from the Dialog Manager. Specify a value of <code>NIL</code> if you have not added any items to the dialog box and want the standard items handled in the standard ways. See “Writing a Dialog Hook Function” on page 4-196 for a description of the dialog hook function.
<code>filterProc</code>	A pointer to your modal-dialog filter function, which determines how the <code>ModalDialog</code> procedure filters events when called by the <code>CustomPutFile</code> procedure. Specify a value of <code>NIL</code> if you are not supplying your own function. See “Writing a Modal-Dialog Filter Function” on page 4-203 for a description of the modal-dialog filter function.
<code>activeList</code>	A pointer to a list of all dialog items that can be activated—that is, can be the target of keyboard input. If you supply an <code>activeList</code> parameter of <code>NIL</code> , <code>CustomPutFile</code> uses the default targets (the filename field and the list of files and folders). If you have added any fields that can accept keyboard input, you must modify the list. The list is stored as an array of 16-bit integers. The first integer is the number of items in the list. The remaining integers are the item numbers of all possible keyboard targets, in the order that they are activated by the Tab key.

Standard File Package

`activateProc`

A pointer to your activation procedure, which controls the highlighting of dialog items that are defined by your application and that can receive keyboard input. See “Writing an Activation Procedure” on page 4-205 for a description of the activation procedure.

`yourDataPtr`

Any 4-byte value; usually, a pointer to optional data supplied by your application. When `CustomPutFile` calls any of your callback routines, it adds this parameter, making the data available to your callback routines. If you are not supplying any data of your own, you can specify a value of `NIL`.

DESCRIPTION

The `CustomPutFile` procedure is an alternative to `StandardPutFile` when you want to display a customized Save dialog box or handle the default dialog box in a customized way. During the dialog, `CustomPutFile` gets and handles events (possibly with the assistance of application-defined callback routines) until the user completes the interaction, either by selecting a name and authorizing the save operation or by canceling the save operation. The `CustomPutFile` procedure returns the user’s input in a record of type `StandardFileReply`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `CustomPutFile` are

Trap macro	Selector
<code>_Pack3</code>	<code>\$0007</code>

SPECIAL CONSIDERATIONS

The `CustomPutFile` procedure is not available in all versions of system software. Use the `Gestalt` function to determine whether `CustomPutFile` is available before calling it.

Because `CustomPutFile` may move memory, you should not call it at interrupt time.

SFPutFile

Use the `SFPutFile` procedure to display the standard Save dialog box when the user is saving a file.

```
PROCEDURE SFPutFile (where: Point; prompt: Str255;
                    origName: Str255; dlgHook: DlgHookProcPtr;
                    VAR reply: SReply);
```

Standard File Package

where	The upper-left corner of the dialog box, in global coordinates.
prompt	The prompt message to be displayed over the text field.
origName	The initial name of the file.
dlgHook	A pointer to your dialog hook function, which handles item selections received from the Dialog Manager. Specify a value of <code>NIL</code> if you want the standard items handled in the standard ways. See “Writing a Dialog Hook Function” on page 4-196 for a description of the dialog hook function.
reply	The reply record, which <code>SFPutFile</code> fills in before returning.

DESCRIPTION

The `SFPutFile` procedure presents a dialog box through which the user specifies the name and location of a file to be written to. During the dialog, `SFPutFile` gets and handles events until the user completes the interaction, either by selecting a name and authorizing the save or by canceling the save. The `SFPutFile` procedure returns the user’s input in a record of type `SFReply`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SFPutFile` are

Trap macro	Selector
<code>_Pack3</code>	<code>\$0001</code>

SPECIAL CONSIDERATIONS

Because `SFPutFile` may move memory, you should not call it at interrupt time.

SFPPutFile

Use the `SFPPutFile` procedure when your application requires more control over the Save dialog box than is possible using `SFPutFile`.

```
PROCEDURE SFPPutFile (where: Point; prompt: Str255;
                     origName: Str255; dlgHook: DlgHookProcPtr;
                     VAR reply: SFReply; dlgID: Integer;
                     filterProc: ModalFilterProcPtr);
```

where	The upper-left corner of the dialog box, in global coordinates.
prompt	The prompt message to be displayed over the text field.
origName	The initial name of the file, if any.

Standard File Package

<code>dlgHook</code>	A pointer to your dialog hook function, which handles item selections received from the Dialog Manager. Specify a value of <code>NIL</code> if you have not added any items to the dialog box and want the standard items handled in the standard ways. See “Writing a Dialog Hook Function” on page 4-196 for a description of the dialog hook function.
<code>reply</code>	The reply record, which <code>SFPPutFile</code> fills in before returning.
<code>dlgID</code>	The resource ID of a customized dialog template. To use the standard template, set this parameter to <code>-3999</code> .
<code>filterProc</code>	A pointer to your modal-dialog filter function, which determines how the <code>ModalDialog</code> procedure filters events when called by the <code>SFPPutFile</code> procedure. Specify a value of <code>NIL</code> if you are not supplying your own function. See “Writing a Modal-Dialog Filter Function” on page 4-203 for a description of the modal-dialog filter function.

DESCRIPTION

The `SFPPutFile` procedure is an alternative to `SFPutFile` when you want to display a customized Save dialog box or handle the default dialog box in a customized way. During the dialog, `SFPPutFile` gets and handles events (possibly with the assistance of application-defined callback routines) until the user completes the interaction, either by selecting a name and authorizing the save operation or by canceling the save operation. `SFPPutFile` returns the user’s input in a record of type `SFReply`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SFPPutFile` are

Trap macro	Selector
<code>_Pack3</code>	<code>\$0003</code>

SPECIAL CONSIDERATIONS

Because `SFPPutFile` may move memory, you should not call it at interrupt time.

Opening Files

You can use the `StandardGetFile` procedure to present the standard user interface when the user asks to open a file. If you need to add elements to the default dialog boxes or exercise greater control over user actions in the dialog box, use `CustomGetFile`.

If your application is designed to execute in system software versions earlier than version 7.0, you can use the corresponding procedures `SFGetFile` and `SFPGetFile`.

StandardGetFile

You can use the `StandardGetFile` procedure to display the default Open dialog box when the user is opening a file.

```
PROCEDURE StandardGetFile (fileFilter: FileFilterProcPtr;
                           numTypes: Integer;
                           typeList: SFTYPEList;
                           VAR reply: StandardFileReply);
```

<code>fileFilter</code>	A pointer to an optional file filter function, provided by your application, through which <code>StandardGetFile</code> passes files of the specified types.
<code>numTypes</code>	The number of file types to be displayed. If you specify a <code>numTypes</code> value of -1, the first filtering passes files of all types.
<code>typeList</code>	A list of file types to be displayed.
<code>reply</code>	The reply record, which <code>StandardGetFile</code> fills in before returning.

DESCRIPTION

The `StandardGetFile` procedure presents a dialog box through which the user specifies the name and location of a file to be opened. While the dialog box is active, `StandardGetFile` gets and handles events until the user completes the interaction, either by selecting a file to open or by canceling the operation. `StandardGetFile` returns the user's input in a record of type `StandardFileReply`.

The `fileFilter`, `numTypes`, and `typeList` parameters together determine which files appear in the displayed list. The first filtering is by file type, which you specify in the `numTypes` and `typeList` parameters. The `numTypes` parameter specifies the number of file types to be displayed. You can specify one or more types. If you specify a `numTypes` value of -1, the first filtering passes files of all types.

The `fileFilter` parameter points to an optional file filter function, provided by your application, through which `StandardGetFile` passes files of the specified types. See "Writing a File Filter Function" on page 4-195 for a description of the file filter function.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `StandardGetFile` are

Trap macro	Selector
<code>_Pack3</code>	<code>\$0006</code>

SPECIAL CONSIDERATIONS

The `StandardGetFile` procedure is not available in all versions of system software. Use the `Gestalt` function to determine whether `StandardGetFile` is available before calling it.

Because `StandardGetFile` may move memory, you should not call it at interrupt time.

CustomGetFile

Call the `CustomGetFile` procedure when your application requires more control over the Open dialog box than is possible using `StandardGetFile`.

```
PROCEDURE CustomGetFile (fileFilter: FileFilterYDProcPtr;
                        numTypes: Integer;
                        typeList: SFTypeList;
                        VAR reply: StandardFileReply;
                        dlgID: Integer;
                        where: Point;
                        dlgHook: DlgHookYDProcPtr;
                        filterProc: ModalFilterYDProcPtr;
                        activeList: Ptr;
                        activateProc: ActivateYDProcPtr;
                        yourDataPtr: UNIV Ptr);
```

- `fileFilter` A pointer to an optional file filter function, provided by your application, through which `CustomGetFile` passes files of the specified types.
- `numTypes` The number of file types to be displayed. If you specify a `numTypes` value of -1, the first filtering passes files of all types.
- `typeList` A list of file types to be displayed.
- `reply` The reply record, which `CustomGetFile` fills in before returning.
- `dlgID` The resource ID of a customized dialog template. To use the standard template, set this parameter to 0.
- `where` The upper-left corner of the dialog box in global coordinates. If you specify the point (-1,-1), `CustomGetFile` automatically centers the dialog box on the screen.
- `dlgHook` A pointer to your dialog hook function, which handles item selections received from the Dialog Manager. Specify a value of `NIL` if you have not added any items to the dialog box and want the standard items handled in the standard ways. See “Writing a Dialog Hook Function” on page 4-196 for a description of the dialog hook function.
- `filterProc` A pointer to your modal-dialog filter function, which determines how `ModalDialog` filters events when called by `CustomGetFile`. Specify a value of `NIL` if you are not supplying your own function. See “Writing a Modal-Dialog Filter Function” on page 4-203 for a description of the modal-dialog filter function.
- `activeList` A pointer to a list of all dialog items that can be activated—that is, made the target of keyboard input. The list is stored as an array of 16-bit integers. The first integer is the number of items in the list. The remaining integers are the item numbers of all possible keyboard targets, in the order that they are activated by the Tab key. If you supply an `activeList` parameter of `NIL`, `CustomGetFile` directs all keyboard input to the displayed list.

Standard File Package

`activateProc`

A pointer to your activation procedure, which controls the highlighting of dialog items that are defined by your application and that can receive keyboard input. See “Writing an Activation Procedure” on page 4-205 for a description of the activation procedure.

`yourDataPtr`

A pointer to optional data supplied by your application. When `CustomGetFile` calls any of your callback routines, it pushes this parameter on the stack, making the data available to your callback routines. If you are not supplying any data of your own, specify a value of `NIL`.

DESCRIPTION

The `CustomGetFile` procedure is an alternative to `StandardGetFile` when you want to use a customized dialog box or handle the default Open dialog box in a customized way. `CustomGetFile` presents a dialog box through which the user specifies the name and location of a file to be opened. While the dialog box is active, `CustomGetFile` gets and handles events until the user completes the interaction, either by selecting a file to open or by canceling the operation. `CustomGetFile` returns the user’s input in a record of type `StandardFileReply`.

The first four parameters are similar to the same parameters in `StandardGetFile`. The `fileFilter`, `numTypes`, and `typeList` parameters determine which files appear in the list of choices. If you specify a value of -1 in the `numTypes` parameter, `CustomGetFile` displays or passes to your file filter function all files and folders (not just the files) at the current level of the display hierarchy. If you provide a filter function, `CustomGetFile` passes it both the pointer to the catalog entry for each file to be processed and also a pointer to the optional data passed by your application in its call to `CustomGetFile`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `CustomGetFile` are

Trap macro	Selector
<code>_Pack3</code>	<code>\$0008</code>

SPECIAL CONSIDERATIONS

The `CustomGetFile` procedure is not available in all versions of system software. Use the `Gestalt` function to determine whether `CustomGetFile` is available before calling it.

Because `CustomGetFile` may move memory, you should not call it at interrupt time.

SFGetFile

Use the `SFGetFile` procedure to display the default Open dialog box when the user is opening a file.

```
PROCEDURE SFGetFile (where: Point; prompt: Str255;
                    fileFilter: FileFilterProcPtr;
                    numTypes: Integer; typeList: SFTypelist;
                    dlgHook: DlgHookProcPtr; VAR reply: SFReply);
```

<code>where</code>	The upper-left corner of the dialog box, in global coordinates.
<code>prompt</code>	Ignored.
<code>fileFilter</code>	A pointer to an optional file filter function, provided by your application, through which <code>SFGetFile</code> passes files of the specified types.
<code>numTypes</code>	The number of file types to be displayed. If you specify a <code>numTypes</code> value of -1, the first filtering passes files of all types.
<code>typeList</code>	A list of file types to be displayed.
<code>dlgHook</code>	A pointer to your dialog hook function, which handles item selections received from the Dialog Manager. Specify a value of <code>NIL</code> if you want the standard items handled in the standard ways.
<code>reply</code>	The reply record, which <code>SFGetFile</code> fills in before returning.

DESCRIPTION

The `SFGetFile` procedure displays a dialog box listing the names of a specific group of files from which the user can select one to be opened (as during an Open menu command). During the dialog, `SFGetFile` gets and handles events (possibly with the assistance of application-defined callback routines) until the user completes the interaction, either by selecting a file to open or by canceling the open operation. `SFGetFile` returns the user's input in a record of type `SFReply`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SFGetFile` are

Trap macro	Selector
<code>_Pack3</code>	<code>\$0002</code>

SPECIAL CONSIDERATIONS

Because `SFGetFile` may move memory, you should not call it at interrupt time.

SFPGetFile

Call the `SFPGetFile` procedure when your application requires more control over the Open dialog box than is possible using `SFGetFile`.

```
PROCEDURE SFPGetFile (where: Point; prompt: Str255;
                    fileFilter: FileFilterProcPtr;
                    numTypes: Integer; typeList: SFTypeList;
                    dlgHook: DlgHookProcPtr;
                    VAR reply: SFReply; dlgID: Integer;
                    filterProc: ModalFilterProcPtr);
```

<code>where</code>	The upper-left corner of the dialog box, in global coordinates.
<code>prompt</code>	Ignored.
<code>fileFilter</code>	A pointer to an optional file filter function, provided by your application, through which <code>SFPGetFile</code> passes files of the specified types.
<code>numTypes</code>	The number of file types to be displayed. If you specify a <code>numTypes</code> value of -1, the first filtering passes files of all types.
<code>typeList</code>	A list of file types to be displayed.
<code>dlgHook</code>	A pointer to your dialog hook function, which handles item selections received from the Dialog Manager. Specify a value of <code>NIL</code> if you have not added any items to the dialog box and want the standard items handled in the standard ways.
<code>reply</code>	The reply record, which <code>SFPGetFile</code> fills in before returning.
<code>dlgID</code>	The resource ID of a customized dialog template.
<code>filterProc</code>	A pointer to your modal-dialog filter function, which determines how the <code>ModalDialog</code> procedure filters events when called by the <code>SFPGetFile</code> procedure. Specify a value of <code>NIL</code> if you are not supplying your own function.

DESCRIPTION

The `SFPGetFile` procedure is an alternative to `SFGetFile` when you want to display a customized Open dialog box or handle the default dialog box in a customized way. During the dialog, `SFPGetFile` gets and handles events (possibly with the assistance of application-defined callback routines) until the user completes the interaction, either by selecting a file to open or by canceling the open operation. `SFPGetFile` returns the user's input in a record of type `SFReply`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SFPGetFile` are

Trap macro	Selector
<code>_Pack3</code>	<code>\$0004</code>

SPECIAL CONSIDERATIONS

Because `SFPGetFile` may move memory, you should not call it at interrupt time.

Application-Defined Routines

This section describes the application-defined routines whose addresses you pass to some of the Standard File Package routines. You can define

- a file filter function for determining which files the user can open
- a dialog hook function for handling user actions in the dialog boxes
- a modal-dialog filter function for handling user events received from the Event Manager
- an activation procedure for highlighting the display when keyboard input is directed at a customized field defined by your application

File Filter Functions

You specify a file filter function to determine which files appear in the displayed list of files and folders when the user is opening a file. You can define a standard or custom file filter.

MyStandardFileFilter

A file filter function whose address is passed to `StandardGetFile` should have the following form:

```
FUNCTION MyStandardFileFilter (pb: CInfoPBPtr): Boolean;
```

`pb` A pointer to a catalog information parameter block.

DESCRIPTION

When `StandardGetFile` is displaying the contents of a volume or folder, it checks the file type of each file and filters out files whose types do not match your application's specifications. If your application also supplies a file filter function, the Standard File Package calls that function each time it identifies a file of an acceptable type.

When your file filter function is called, it is passed, in the `pb` parameter, a pointer to a catalog information parameter block. See the chapter "File Manager" in this book for a description of the fields of this parameter block.

Your function evaluates the catalog information parameter block and returns a Boolean value that determines whether the file is filtered (that is, a value of `TRUE` suppresses display of the filename, and a value of `FALSE` allows the display). If you do not supply a file filter function, the Standard File Package displays all files of the specified types.

SEE ALSO

See "Writing a File Filter Function" on page 4-195 for a sample file filter function.

MyCustomFileFilter

A file filter function whose address is passed to `CustomGetFile` should have the following form:

```
FUNCTION MyCustomFileFilter (pb: CInfoBPTr; myDataPtr: Ptr):
    Boolean;
```

`pb` A pointer to a catalog information parameter block.
`myDataPtr` A pointer to the optional data whose address is passed to
 `CustomGetFile`.

DESCRIPTION

When `CustomGetFile` is displaying the contents of a volume or folder, it checks the file type of each file and filters out files whose types do not match your application's specifications. If your application also supplies a file filter function, the Standard File Package calls that function each time it identifies a file of an acceptable type.

When your file filter function is called, it is passed, in the `pb` parameter, a pointer to a catalog information parameter block. See the chapter "File Manager" in this book for a description of the fields of this parameter block.

Your function evaluates the catalog information parameter block and returns a Boolean value that determines whether the file is filtered (that is, a value of `TRUE` suppresses display of the filename, and a value of `FALSE` allows the display). If you do not supply a file filter function, the Standard File Package displays all files of the specified types.

SEE ALSO

See "Writing a File Filter Function" on page 4-195 for a sample file filter function.

Dialog Hook Functions

A dialog hook function handles user selections in a dialog box.

MyDlgHook

A dialog hook function should have the following form:

```
FUNCTION MyDlgHook (item: Integer; theDialog: DialogPtr;
    myDataPtr: Ptr): Integer;
```

`item` The number of the item selected.
`theDialog` A pointer to the dialog record of the dialog box.

Standard File Package

myDataPtr A pointer to the optional data whose address is passed to CustomGetFile or CustomPutFile.

DESCRIPTION

You supply a dialog hook function to handle user selections of items that you added to a dialog box. If you provide a dialog hook function, CustomPutFile and CustomGetFile call your function immediately after calling ModalDialog. They pass your function the item number returned by ModalDialog, a pointer to the dialog record, and a pointer to the data received from your application, if any.

Your dialog hook function returns as its function result an integer that is either the item number passed to it or some other item number. If your dialog hook function does not handle a selection, it should pass the item number back to the Standard File Package for processing by setting its return value equal to the item number. If your dialog hook function does handle the selection, it should pass back sfHookNullEvent or the number of some other pseudo-item.

SEE ALSO

See “Writing a Dialog Hook Function” on page 4-196 for a sample dialog hook function.

Modal-Dialog Filter Functions

A modal-dialog filter function controls events closer to their source by filtering the events received from the Event Manager. The Standard File Package itself contains an internal modal-dialog filter function that maps keypresses and other user input onto the equivalent dialog box items. If you also want to process events at this level, you can supply your own filter function.

MyModalFilter

A modal-dialog filter function whose address is passed to SFPGetFile or SFPPutFile should have the following form:

```
FUNCTION MyModalFilter (theDialog: DialogPtr;
                       VAR theEvent: EventRecord;
                       VAR itemHit: Integer): Boolean;
```

theDialog A pointer to the dialog record of the dialog box.

theEvent The event record for the event.

itemHit The number of the item selected.

DESCRIPTION

Your modal-dialog filter function determines how the Dialog Manager procedure `ModalDialog` filters events. The `ModalDialog` procedure retrieves events by calling the Event Manager function `GetNextEvent`. The Standard File Package contains an internal filter function that performs some preliminary processing on each event it receives. If you provide a modal-dialog filter function, `ModalDialog` calls your filter function after it calls the internal Standard File Package filter function and before it sends the event to your dialog hook function.

Your modal-dialog filter function returns a Boolean value that reports whether it handled the event. If your function returns a value of `FALSE`, `ModalDialog` processes the event through its own filters. If your function returns a value of `TRUE`, `ModalDialog` returns with no further action.

SEE ALSO

See “Writing a Modal-Dialog Filter Function” on page 4-203 for a sample modal-dialog filter function.

MyModalFilterYD

A modal-dialog filter function whose address is passed to `CustomGetFile` or `CustomPutFile` should have the following form:

```
FUNCTION MyModalFilterYD (theDialog: DialogPtr;
                        VAR theEvent: EventRecord;
                        VAR itemHit: Integer;
                        myDataPtr: Ptr): Boolean;
```

<code>theDialog</code>	A pointer to the dialog record of the dialog box.
<code>theEvent</code>	The event record for the event.
<code>itemHit</code>	The number of the item selected.
<code>myDataPtr</code>	A pointer to the optional data whose address is passed to <code>CustomGetFile</code> or <code>CustomPutFile</code> .

DESCRIPTION

Your modal-dialog filter function determines how the Dialog Manager procedure `ModalDialog` filters events. The `ModalDialog` procedure retrieves events by calling the Event Manager function `GetNextEvent`. The Standard File Package contains an internal filter function that performs some preliminary processing on each event it receives. If you provide a modal-dialog filter function, `ModalDialog` calls your filter function after it calls the internal Standard File Package filter function and before it sends the event to your dialog hook function.

Standard File Package

Your modal-dialog filter function returns a Boolean value that reports whether it handled the event. If your function returns a value of `FALSE`, `ModalDialog` processes the event through its own filters. If your function returns a value of `TRUE`, `ModalDialog` returns with no further action.

SEE ALSO

See “Writing a Modal-Dialog Filter Function” on page 4-203 for a sample modal-dialog filter function.

Activation Procedures

An activation procedure controls the highlighting of dialog items that are defined by your application and can receive keyboard input.

MyActivateProc

An activation procedure should have the following form:

```
PROCEDURE MyActivateProc (theDialog: DialogPtr; itemNo: Integer;
                          activating: Boolean; myDataPtr: Ptr);
```

`theDialog` A pointer to the dialog record of the dialog box.

`itemNo` The number of the item selected.

`activating` A Boolean value that specifies whether the field is being activated (`TRUE`) or deactivated (`FALSE`).

`myDataPtr` A pointer to the optional data whose address is passed to `CustomGetFile` or `CustomPutFile`.

DESCRIPTION

Your activation procedure controls the highlighting of dialog items that are defined by your application and can receive keyboard input. Ordinarily, you need to supply an activation procedure only if your application builds a list from which the user can select entries. The Standard File Package supplies the activation procedure for the file display list and for all `TextEdit` fields. You can also use the activation procedure to keep track of which field is receiving keyboard input, if your application needs that information.

Your application is responsible for removing the highlighting when one of its fields becomes inactive and for adding the highlighting when one of its fields becomes active. The Standard File Package can handle the highlighting of all `TextEdit` fields, even those defined by your application.

Introduction to Memory Management

This chapter is a general introduction to memory management on Macintosh computers. It describes how the Operating System organizes and manages the available memory, and it shows how you can use the services provided by the Memory Manager and other system software components to manage the memory in your application partition effectively.

You should read this chapter if your application or other software allocates memory dynamically during its execution. This chapter describes how to

- set up your application partition at launch time
- determine the amount of free memory in your application heap
- allocate and dispose of blocks of memory in your application heap
- minimize fragmentation in your application heap caused by blocks of memory that cannot move
- implement a scheme to avoid low-memory conditions

You should be able to accomplish most of your application's memory allocation and management by following the instructions given in this chapter. If, however, your application needs to allocate memory outside its own partition (for instance, in the system heap), you need to read the chapter "Memory Manager" in this book. If your application has timing-critical requirements or installs procedures that execute at interrupt time, you need to read the chapter "Virtual Memory Manager" in this book. If your application's executable code is divided into multiple segments, you might also want to look at the chapter "Segment Manager" in *Inside Macintosh: Processes* for guidelines on how to divide your code into segments. If your application uses resources, you need to read the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for information on managing memory allocated to resources.

This chapter begins with a description of how the Macintosh Operating System organizes the available physical random-access memory (RAM) in a Macintosh computer and how it allocates memory to open applications. Then this chapter describes in detail how the

Memory Manager allocates blocks of memory in your application's heap and how to use the routines provided by the Memory Manager to perform the memory-management tasks listed above.

This chapter ends with descriptions of the routines used to perform these tasks. The "Memory Management Reference" and "Result Codes" sections in this chapter are subsets of the corresponding sections in the remaining chapters in this book.

About Memory

A Macintosh computer's available RAM is used by the Operating System, applications, and other software components, such as device drivers and system extensions. This section describes both the general organization of memory by the Operating System and the organization of the memory partition allocated to your application when it is launched. This section also provides a preliminary description of three related memory topics:

- temporary memory
- virtual memory
- 24- and 32-bit addressing

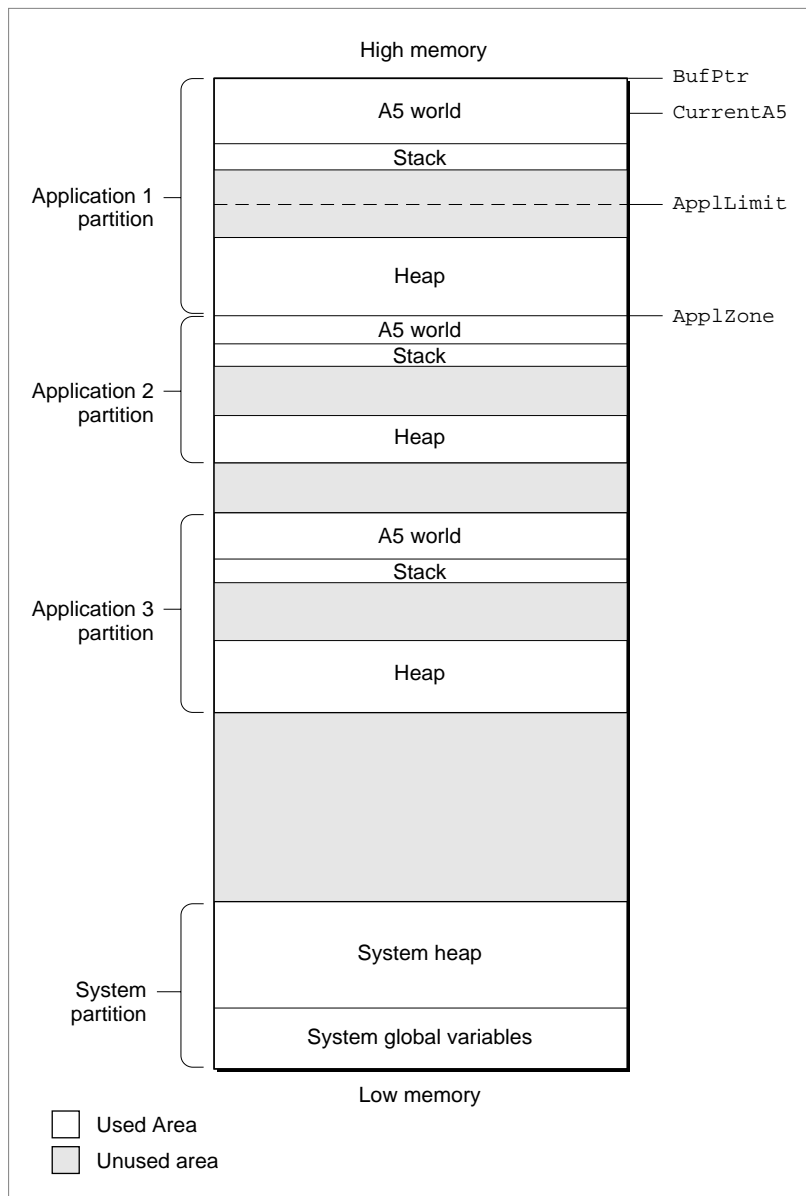
For more complete information on these three topics, you need to read the remaining chapters in this book.

Organization of Memory by the Operating System

When the Macintosh Operating System starts up, it divides the available RAM into two broad sections. It reserves for itself a zone or **partition** of memory known as the **system partition**. The system partition always begins at the lowest addressable byte of memory (memory address 0) and extends upward. The system partition contains a system heap and a set of global variables, described in the next two sections.

All memory outside the system partition is available for allocation to applications or other software components. In system software version 7.0 and later (or when MultiFinder is running in system software versions 5.0 and 6.0), the user can have multiple applications open at once. When an application is launched, the Operating System assigns it a section of memory known as its **application partition**. In general, an application uses only the memory contained in its own application partition.

Figure 5-1 illustrates the organization of memory when several applications are open at the same time. The system partition occupies the lowest position in memory. Application partitions occupy part of the remaining space. Note that application partitions are loaded into the top part of memory first.

Figure 5-1 Memory organization with several applications open

In Figure 5-1, three applications are open, each with its own application partition. The application labeled Application 1 is the active application. (The labels on the right side of the figure are system global variables, explained in “The System Global Variables” on page 5-238.)

The System Heap

The main part of the system partition is an area of memory known as the **system heap**. In general, the system heap is reserved for exclusive use by the Operating System and other system software components, which load into it various items such as system resources, system code segments, and system data structures. All system buffers and queues, for example, are allocated in the system heap.

The system heap is also used for code and other resources that do not belong to specific applications, such as code resources that add features to the Operating System or that provide control of special-purpose peripheral equipment. System patches and system extensions (stored as code resources of type 'INIT') are loaded into the system heap during the system startup process. Hardware device drivers (stored as code resources of type 'DRVr') are loaded into the system heap when the driver is opened.

Most applications don't need to load anything into the system heap. In certain cases, however, you might need to load resources or code segments into the system heap. For example, if you want a vertical retrace task to continue to execute even when your application is in the background, you need to load the task and any data associated with it into the system heap. Otherwise, the Vertical Retrace Manager ignores the task when your application is in the background.

The System Global Variables

The lowest part of memory is occupied by a collection of global variables called **system global variables** (or **low-memory system global variables**). The Operating System uses these variables to maintain different kinds of information about the operating environment. For example, the `Ticks` global variable contains the number of ticks (sixtieths of a second) that have elapsed since the system was most recently started up. Similar variables contain, for example, the height of the menu bar (`MBarHeight`) and pointers to the heads of various operating-system queues (`DTQueue`, `FSQHdr`, `VLQueue`, and so forth). Most low-memory global variables are of this variety: they contain information that is generally useful only to the Operating System or other system software components.

Other low-memory global variables contain information about the current application. For example, the `ApplZone` global variable contains the address of the first byte of the active application's partition. The `ApplLimit` global variable contains the address of the last byte the active application's heap can expand to include. The `CurrentA5` global variable contains the address of the boundary between the active application's global variables and its application parameters. Because these global variables contain information about the active application, the Operating System changes the values of these variables whenever a context switch occurs.

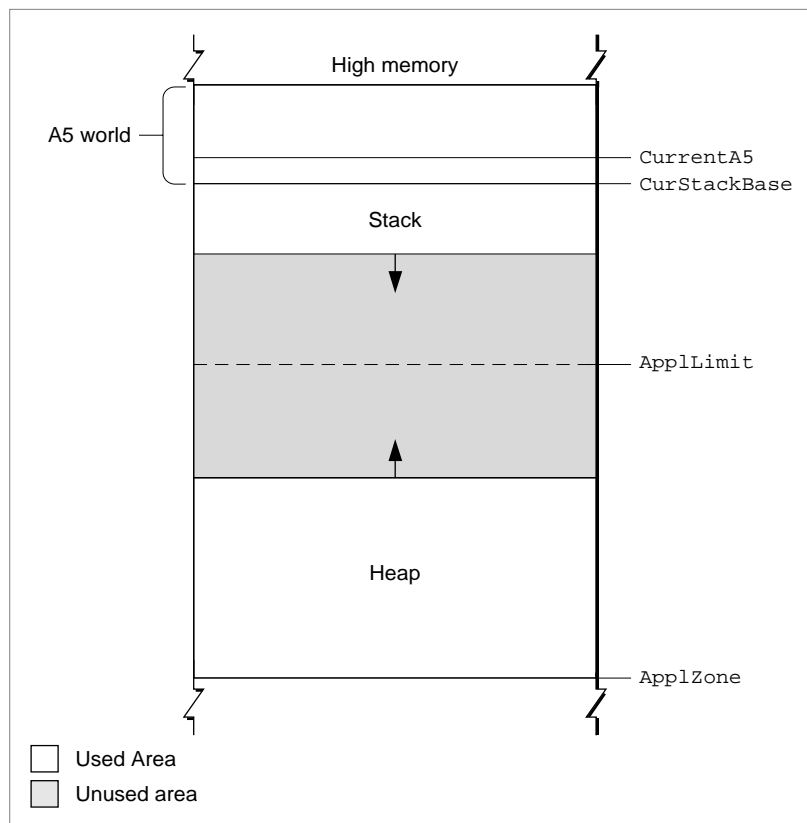
In general, it is best to avoid reading or writing low-memory system global variables. Most of these variables are undocumented, and the results of changing their values can be unpredictable. Usually, when the value of a low-memory global variable is likely to be useful to applications, the system software provides a routine that you can use to read or write that value. For example, you can get the current value of the `Ticks` global variable by calling the `TickCount` function.

In rare instances, there is no routine that reads or writes the value of a documented global variable. In those cases, you might need to read or write that value directly. See the chapter “Memory Manager” in this book for instructions on reading and writing the values of low-memory global variables from a high-level language.

Organization of Memory in an Application Partition

When your application is launched, the Operating System allocates for it a partition of memory called its **application partition**. That partition contains required segments of the application’s code as well as other data associated with the application. Figure 5-2 illustrates the general organization of an application partition.

Figure 5-2 Organization of an application partition



Your application partition is divided into three major parts:

- the application stack
- the application heap
- the application global variables and A5 world

The heap is located at the low-memory end of your application partition and always expands (when necessary) toward high memory. The A5 world is located at the high-memory end of your application partition and is of fixed size. The stack begins at the low-memory end of the A5 world and expands downward, toward the top of the heap.

As you can see in Figure 5-2, there is usually an unused area of memory between the stack and the heap. This unused area provides space for the stack to grow without encroaching upon the space assigned to the application heap. In some cases, however, the stack might grow into space reserved for the application heap. If this happens, it is very likely that data in the heap will become corrupted.

The `ApplLimit` global variable marks the upper limit to which your heap can grow. If you call the `MaxApplZone` procedure at the beginning of your program, the heap immediately extends all the way up to this limit. If you were to use all of the heap's free space, the Memory Manager would not allow you to allocate additional blocks above `ApplLimit`. If you do not call `MaxApplZone`, the heap grows toward `ApplLimit` whenever the Memory Manager finds that there is not enough memory in the heap to fill a request. However, once the heap grows up to `ApplLimit`, it can grow no further. Thus, whether you maximize your application heap or not, you can use only the space between the bottom of the heap and `ApplLimit`.

Unlike the heap, the stack is not bounded by `ApplLimit`. If your application uses heavily nested procedures with many local variables or uses extensive recursion, the stack could grow downward beyond `ApplLimit`. Because you do not use Memory Manager routines to allocate memory on the stack, the Memory Manager cannot stop your stack from growing beyond `ApplLimit` and possibly encroaching upon space reserved for the heap. However, a vertical retrace task checks approximately 60 times each second to see if the stack has moved into the heap. If it has, the task, known as the "stack sniffer," generates a system error. This system error alerts you that you have allowed the stack to grow too far, so that you can make adjustments. See "Changing the Size of the Stack" on page 5-271 for instructions on how to change the size of your application stack.

Note

To ensure during debugging that your application generates this system error if the stack extends beyond `ApplLimit`, you should call `MaxApplZone` at the beginning of your program to expand the heap to `ApplLimit`. For more information on expanding the heap, see "Setting Up the Application Heap" beginning on page 5-270. ♦

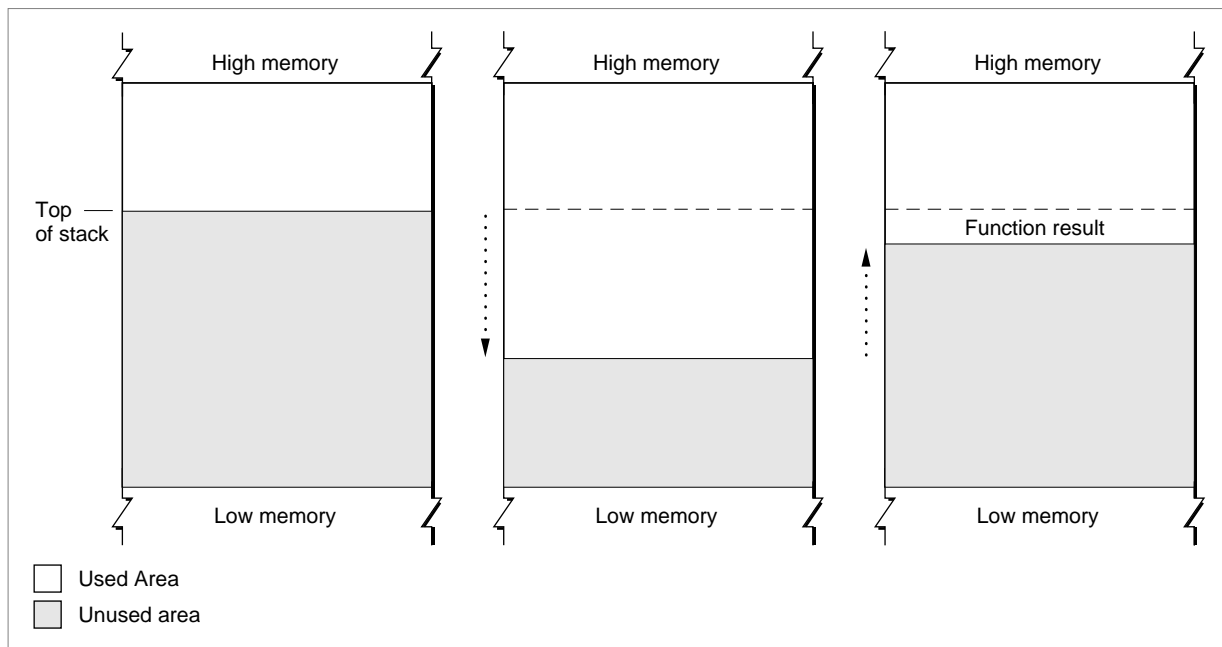
The Application Stack

The **stack** is an area of memory in your application partition that can grow or shrink at one end while the other end remains fixed. This means that space on the stack is always allocated and released in LIFO (last-in, first-out) order. The last item allocated is always the first to be released. It also means that the allocated area of the stack is always contiguous. Space is released only at the top of the stack, never in the middle, so there can never be any unallocated "holes" in the stack.

By convention, the stack grows from high memory toward low memory addresses. The end of the stack that grows or shrinks is usually referred to as the “top” of the stack, even though it’s actually at the lower end of memory occupied by the stack.

Because of its LIFO nature, the stack is especially useful for memory allocation connected with the execution of functions or procedures. When your application calls a routine, space is automatically allocated on the stack for a stack frame. A **stack frame** contains the routine’s parameters, local variables, and return address. Figure 5-3 illustrates how the stack expands and shrinks during a function call. The leftmost diagram shows the stack just before the function is called. The middle diagram shows the stack expanded to hold the stack frame. Once the function is executed, the local variables and function parameters are popped off the stack. If the function is a Pascal function, all that remains is the previous stack with the function result on top.

Figure 5-3 The application stack



Note

Dynamic memory allocation on the stack is usually handled automatically if you are using a high-level development language such as Pascal. The compiler generates the code that creates and deletes stack frames for each function or procedure call. ♦

The Application Heap

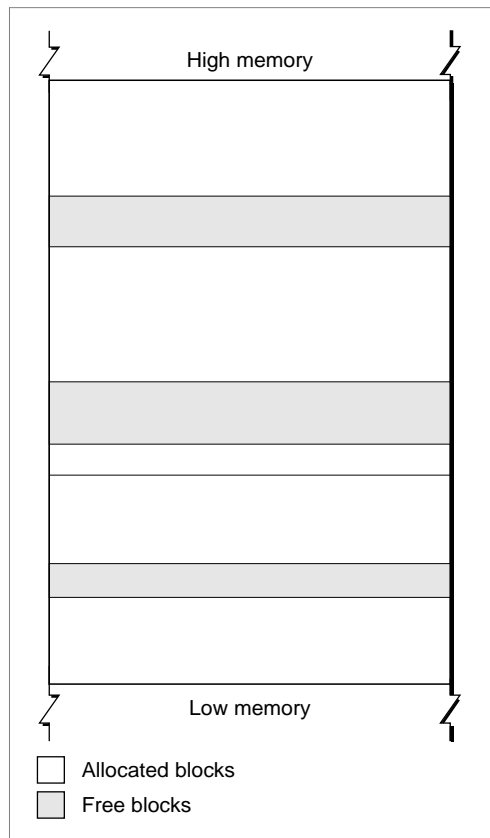
An **application heap** is the area of memory in your application partition in which space is dynamically allocated and released on demand. The heap begins at the low-memory

end of your application partition and extends upward in memory. The heap contains virtually all items that are not allocated on the stack. For instance, your application heap contains the application's code segments and resources that are currently loaded into memory. The heap also contains other dynamically allocated items such as window records, dialog records, document data, and so forth.

You allocate space within your application's heap by making calls to the Memory Manager, either directly (for instance, using the `NewHandle` function) or indirectly (for instance, using a routine such as `NewWindow`, which calls Memory Manager routines). Space in the heap is allocated in **blocks**, which can be of any size needed for a particular object.

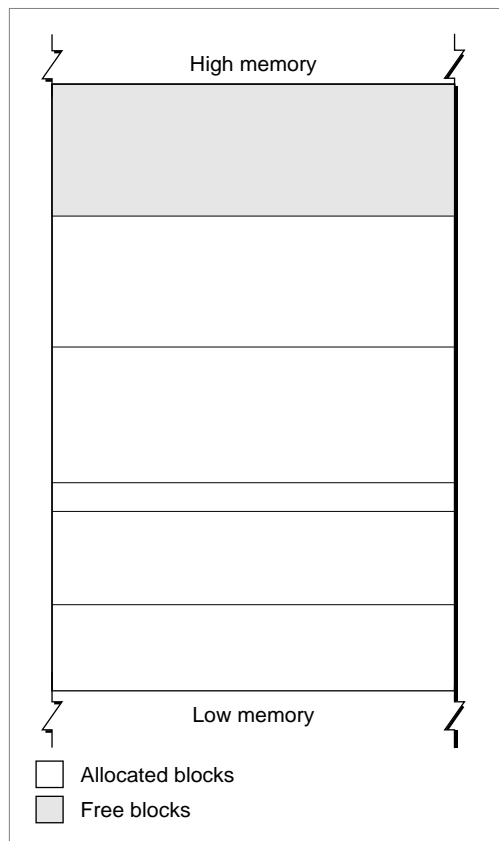
The Memory Manager does all the necessary housekeeping to keep track of blocks in the heap as they are allocated and released. Because these operations can occur in any order, the heap doesn't usually grow and shrink in an orderly way, as the stack does. Instead, after your application has been running for a while, the heap can tend to become fragmented into a patchwork of allocated and free blocks, as shown in Figure 5-4. This fragmentation is known as **heap fragmentation**.

Figure 5-4 A fragmented heap



One result of heap fragmentation is that the Memory Manager might not be able to satisfy your application's request to allocate a block of a particular size. Even though there is enough free space available, the space is broken up into blocks smaller than the requested size. When this happens, the Memory Manager tries to create the needed space by moving allocated blocks together, thus collecting the free space in a single larger block. This operation is known as **heap compaction**. Figure 5-5 shows the results of compacting the fragmented heap shown in Figure 5-4.

Figure 5-5 A compacted heap



Heap fragmentation is generally not a problem as long as the blocks of memory you allocate are free to move during heap compaction. There are, however, two situations in which a block is not free to move: when it is a nonrelocatable block, and when it is a locked, relocatable block. To minimize heap fragmentation, you should use nonrelocatable blocks sparingly, and you should lock relocatable blocks only when absolutely necessary. See “Relocatable and Nonrelocatable Blocks” starting on page 5-248 for a description of relocatable and nonrelocatable blocks, and “Heap Fragmentation” on page 5-256 for a description of how best to avoid fragmenting your heap.

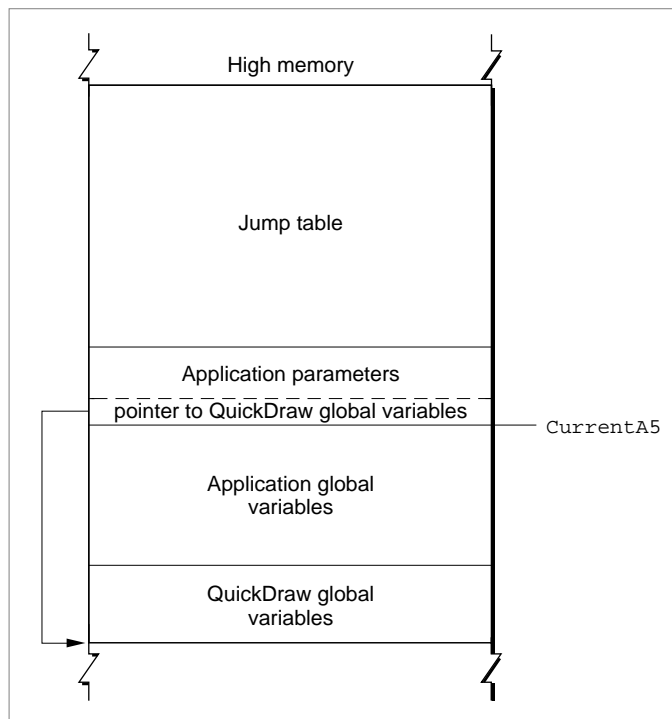
The Application Global Variables and A5 World

Your application's global variables are stored in an area of memory near the top of your application partition known as the application **A5 world**. The A5 world contains four kinds of data:

- application global variables
- application QuickDraw global variables
- application parameters
- the application's jump table

Each of these items is of fixed size, although the sizes of the global variables and of the jump table may vary from application to application. Figure 5-6 shows the standard organization of the A5 world.

Figure 5-6 Organization of an application's A5 world



Note

An application's global variables may appear either above or below the QuickDraw global variables. The relative locations of these two items are determined by your development system's linker. In addition, part of the jump table might appear below the boundary pointed to by CurrentA5. ♦

The system global variable `CurrentA5` points to the boundary between the current application's global variables and its application parameters. For this reason, the application's global variables are found as negative offsets from the value of `CurrentA5`. This boundary is important because the Operating System uses it to access the following information from your application: its global variables, its QuickDraw global variables, the application parameters, and the jump table. This information is known collectively as the A5 world because the Operating System uses the microprocessor's A5 register to point to that boundary.

Your application's **QuickDraw global variables** contain information about its drawing environment. For example, among these variables is a pointer to the current graphics port.

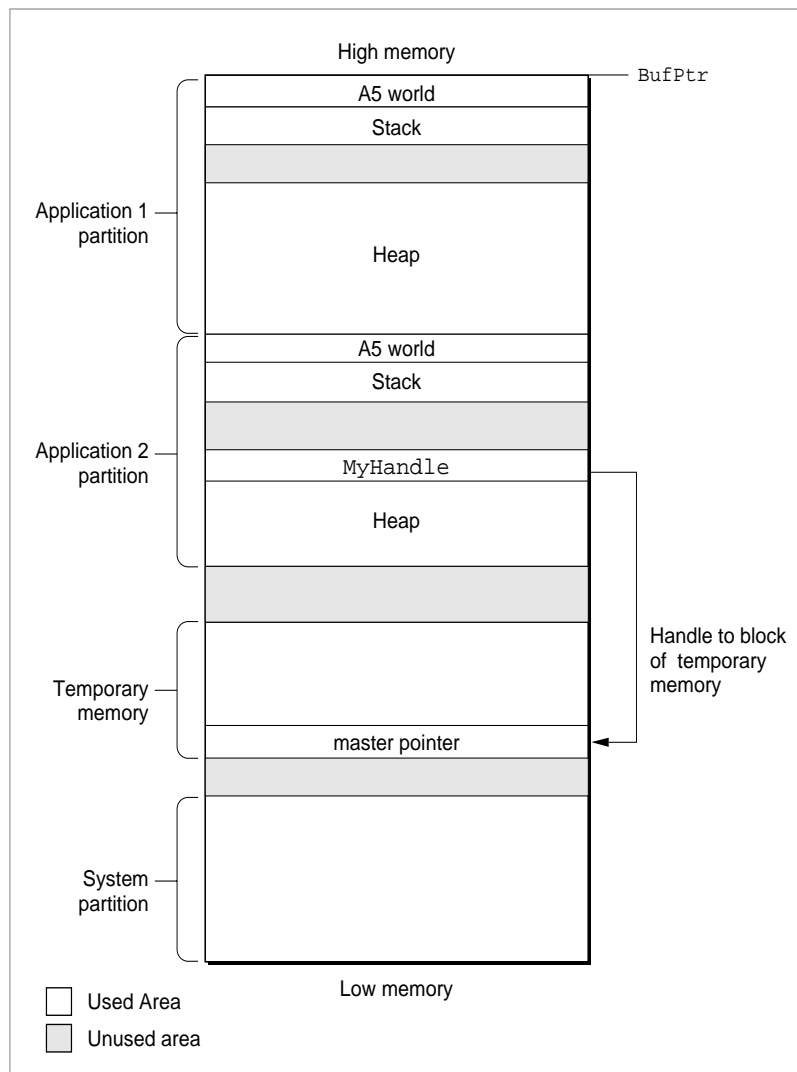
Your application's **jump table** contains an entry for each of your application's routines that is called by code in another segment. The Segment Manager uses the jump table to determine the address of any externally referenced routines called by a code segment. For more information on jump tables, see the chapter "Segment Manager" in *Inside Macintosh: Processes*.

The **application parameters** are 32 bytes of memory located above the application global variables; they're reserved for use by the Operating System. The first long word of those parameters is a pointer to your application's QuickDraw global variables.

Temporary Memory

In the Macintosh multitasking environment, each application is limited to a particular memory partition (whose size is determined by information in the 'SIZE' resource of that application). The size of your application's partition places certain limits on the size of your application heap and hence on the sizes of the buffers and other data structures that your application uses. In general, you specify an application partition size that is large enough to hold all the buffers, resources, and other data that your application is likely to need during its execution.

If for some reason you need more memory than is currently available in your application heap, you can ask the Operating System to let you use any available memory that is not yet allocated to any other application. This memory, known as **temporary memory**, is allocated from the available unused RAM; usually, that memory is not contiguous with the memory in your application's zone. Figure 5-7 shows an application using some temporary memory.

Figure 5-7 Using temporary memory allocated from unused RAM

In Figure 5-7, Application 1 has almost exhausted its application heap. As a result, it has requested and received a large block of temporary memory, extending from the top of Application 2's partition to the top of the allocatable space. Application 1 can use the temporary memory in whatever manner it desires.

Your application should use temporary memory only for occasional short-term purposes that could be accomplished in less space, though perhaps less efficiently. For example, if you want to copy a large file, you might try to allocate a fairly large buffer of temporary memory. If you receive the temporary memory, you can copy data from the source file into the destination file using the large buffer. If, however, the request for temporary memory fails, you can instead use a smaller buffer within your application heap.

Although using the smaller buffer might prolong the copying operation, the file is nonetheless copied.

One good reason for using temporary memory only occasionally is that you cannot assume that you will always receive the temporary memory you request. For example, in Figure 5-7, all the available memory is allocated to the two open applications; any further requests by either one for some temporary memory would fail. For complete details on using temporary memory, see the chapter “Memory Manager” in this book.

Virtual Memory

In system software version 7.0 and later, suitably equipped Macintosh computers can take advantage of a feature of the Operating System known as **virtual memory**, by which the machines have a logical address space that extends beyond the limits of the available physical memory. Because of virtual memory, a user can load more programs and data into the logical address space than would fit in the computer’s physical RAM.

The Operating System extends the address space by using part of the available secondary storage (that is, part of a hard disk) to hold portions of applications and data that are not currently needed in RAM. When some of those portions of memory are needed, the Operating System swaps out unneeded parts of applications or data to the secondary storage, thereby making room for the parts that are needed.

It is important to realize that virtual memory operates transparently to most applications. Unless your application has time-critical needs that might be adversely affected by the operation of virtual memory or installs routines that execute at interrupt time, you do not need to know whether virtual memory is operating. For complete details on virtual memory, see the chapter “Virtual Memory Manager” later in this book.

Addressing Modes

On suitably equipped Macintosh computers, the Operating System supports **32-bit addressing**, that is, the ability to use 32 bits to determine memory addresses. Earlier versions of system software use 24-bit addressing, where the upper 8 bits of memory addresses are ignored or used as flag bits. In a 24-bit addressing scheme, the logical address space has a size of 16 MB. Because 8 MB of this total are reserved for I/O space, ROM, and slot space, the largest contiguous program address space is 8 MB. When 32-bit addressing is in operation, the maximum program address space is 1 GB.

The ability to operate with 32-bit addressing is available only on certain Macintosh models, namely those with systems that contain a 32-bit Memory Manager. (For compatibility reasons, these systems also contain a 24-bit Memory Manager.) In order for your application to work when the machine is using 32-bit addressing, it must be **32-bit clean**, that is, able to run in an environment where all 32 bits of a memory address are significant. Fortunately, writing applications that are 32-bit clean is relatively easy if you follow the guidelines in *Inside Macintosh*. In general, applications are not 32-bit clean because they manipulate flag bits in master pointers directly (for instance, to mark the associated memory blocks as locked or purgeable) instead of using Memory Manager

routines to achieve the desired result. See “Relocatable and Nonrelocatable Blocks” on page 5-248 for a description of master pointers.

▲ **WARNING**

You should never make assumptions about the contents of Memory Manager data structures, including master pointers and zone headers. These structures have changed in the past and they are likely to change again in the future. ▲

Occasionally, an application running when 24-bit addressing is enabled might need to modify memory addresses to make them compatible with the 24-bit Memory Manager. In addition, drivers or other code might need to use 32-bit addresses, even when running in 24-bit mode. See the descriptions of the routines `StripAddress` and `Translate24to32` in the chapter “Memory Management Utilities” for details.

Heap Management

Applications allocate and manipulate memory primarily in their application heap. As you have seen, space in the application heap is allocated and released on demand. When the blocks in your heap are free to move, the Memory Manager can often reorganize the heap to free space when necessary to fulfill a memory-allocation request. In some cases, however, blocks in your heap cannot move. In these cases, you need to pay close attention to memory allocation and management to avoid fragmenting your heap and running out of memory.

This section provides a general description of how to manage blocks of memory in your application heap. It describes

- relocatable and nonrelocatable blocks
- properties of relocatable blocks
- heap purging and compaction
- heap fragmentation
- dangling pointers
- low-memory conditions

For examples of specific techniques you can use to implement the strategies discussed in this section, see “Using Memory” beginning on page 5-270.

Relocatable and Nonrelocatable Blocks

You can use the Memory Manager to allocate two different types of blocks in your heap: nonrelocatable blocks and relocatable blocks. A **nonrelocatable block** is a block of memory whose location in the heap is fixed. In contrast, a **relocatable block** is a block of memory that can be moved within the heap (perhaps during heap compaction).

The Memory Manager sometimes moves relocatable blocks during memory operations so that it can use the space in the heap optimally.

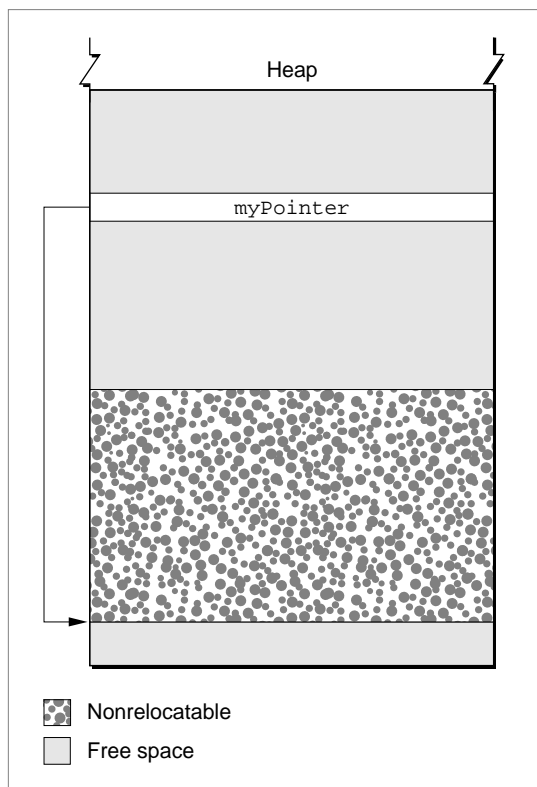
The Memory Manager provides data types that reference both relocatable and nonrelocatable blocks. It also provides routines that allow you to allocate and release blocks of both types.

To reference a nonrelocatable block, you can use a **pointer** variable, defined by the `Ptr` data type.

```
TYPE
    SignedByte    = -128..127;
    Ptr           = ^SignedByte;
```

A pointer is simply the address of an arbitrary byte in memory, and a pointer to a nonrelocatable block of memory is simply the address of the first byte in the block, as illustrated in Figure 5-8. After you allocate a nonrelocatable block, you can make copies of the pointer variable. Because a pointer is the address of a block of memory that cannot be moved, all copies of the pointer correctly reference the block as long as you don't dispose of it.

Figure 5-8 A pointer to a nonrelocatable block



Introduction to Memory Management

The pointer variable itself occupies 4 bytes of space in your application partition. Often the pointer variable is a global variable and is therefore contained in your application's A5 world. But the pointer can also be allocated on the stack or in the heap itself.

To reference relocatable blocks, the Memory Manager uses a scheme known as **double indirection**. The Memory Manager keeps track of a relocatable block internally with a **master pointer**, which itself is part of a nonrelocatable **master pointer block** in your application heap and can never move.

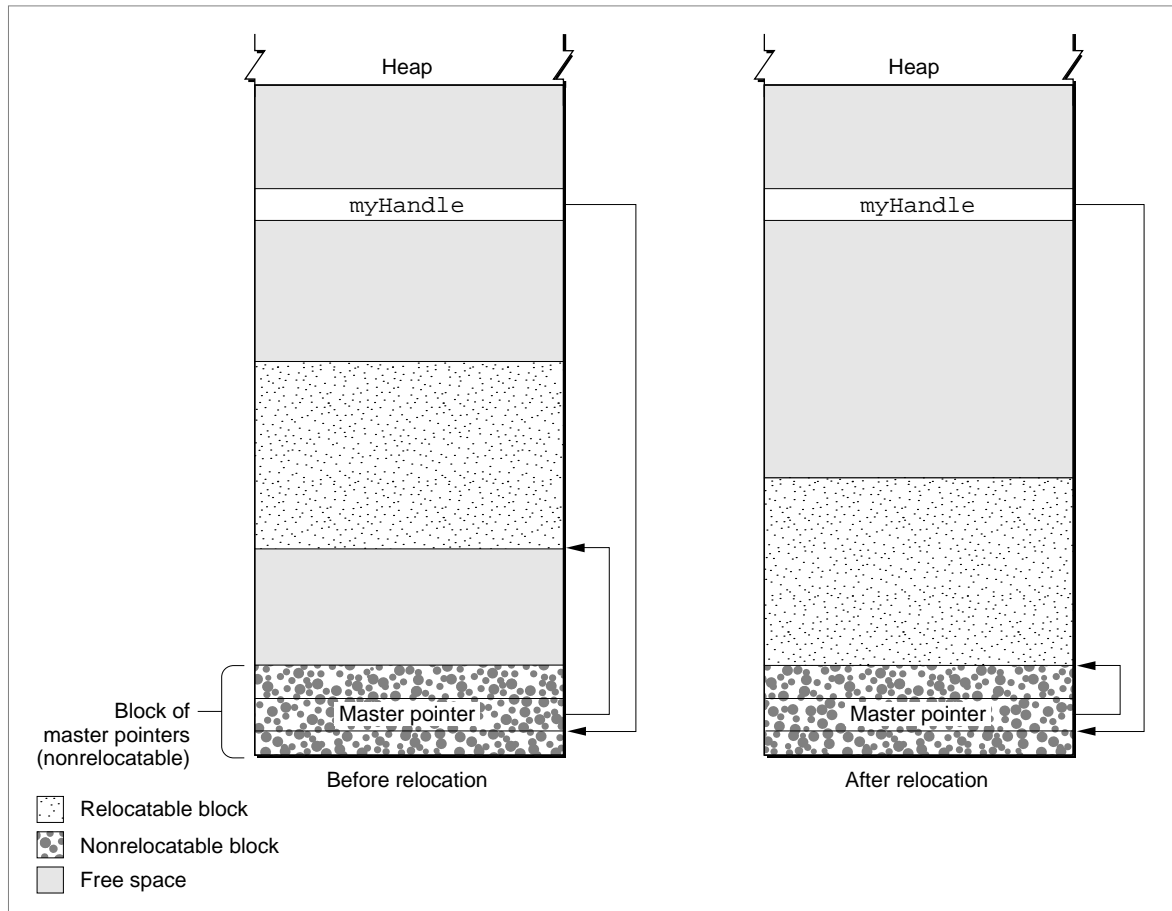
Note

The Memory Manager allocates one master pointer block (containing 64 master pointers) for your application at launch time, and you can call the `MoreMasters` procedure to request that additional master pointer blocks be allocated. See “Setting Up the Application Heap” beginning on page 5-270 for instructions on allocating master pointer blocks. ♦

When the Memory Manager moves a relocatable block, it updates the master pointer so that it always contains the address of the relocatable block. You reference the block with a **handle**, defined by the `Handle` data type.

```
TYPE
    Handle          = ^Ptr;
```

A handle contains the address of a master pointer. The left side of Figure 5-9 shows a handle to a relocatable block of memory located in the middle of the application heap. If necessary (perhaps to make room for another block of memory), the Memory Manager can move that block down in the heap, as shown in the right side of Figure 5-9.

Figure 5-9 A handle to a relocatable block

Master pointers for relocatable objects in your heap are always allocated in your application heap. Because the blocks of master pointers are nonrelocatable, it is best to allocate them as low in your heap as possible. You can do this by calling the `MoreMasters` procedure when your application starts up.

Whenever possible, you should allocate memory in relocatable blocks. This gives the Memory Manager the greatest freedom when rearranging the blocks in your application heap to create a new block of free memory. In some cases, however, you may be forced to allocate a nonrelocatable block of memory. When you call the Window Manager function `NewWindow`, for example, the Window Manager internally calls the `NewPtr` function to allocate a new nonrelocatable block in your application partition. You need to exercise care when calling Toolbox routines that allocate such blocks, lest your application heap become overly fragmented. See “Allocating Blocks of Memory” on page 5-276 for specific guidelines on allocating nonrelocatable blocks.

Using relocatable blocks makes the Memory Manager more efficient at managing available space, but it does carry some overhead. As you have seen, the Memory Manager must allocate extra memory to hold master pointers for relocatable blocks. It groups these master pointers into nonrelocatable blocks. For large relocatable blocks, this extra space is negligible, but if you allocate many very small relocatable blocks, the cost can be considerable. For this reason, you should avoid allocating a very large number of handles to small blocks; instead, allocate a single large block and use it as an array to hold the data you need.

Properties of Relocatable Blocks

As you have seen, a heap block can be either relocatable or nonrelocatable. The designation of a block as relocatable or nonrelocatable is a permanent property of that block. If relocatable, a block can be either locked or unlocked; if it's unlocked, a block can be either purgeable or unpurgeable. These attributes of relocatable blocks can be set and changed as necessary. The following sections explain how to lock and unlock blocks, and how to mark them as purgeable or unpurgeable.

Locking and Unlocking Relocatable Blocks

Occasionally, you might need a relocatable block of memory to stay in one place. To prevent a block from moving, you can **lock** it, using the `HLock` procedure. Once you have locked a block, it won't move. Later, you can **unlock** it, using the `HUnlock` procedure, allowing it to move again.

In general, you need to lock a relocatable block only if there is some danger that it might be moved during the time that you read or write the data in that block. This might happen, for instance, if you dereference a handle to obtain a pointer to the data and (for increased speed) use the pointer within a loop that calls routines that might cause memory to be moved. If, within the loop, the block whose data you are accessing is in fact moved, then the pointer no longer points to that data; this pointer is said to dangle.

Note

Locking a block is only one way to prevent a dangling pointer. See "Dangling Pointers" on page 5-261 for a complete discussion of how to avoid dangling pointers. ♦

Using locked relocatable blocks can, however, slow the Memory Manager down as much as using nonrelocatable blocks. The Memory Manager can't move locked blocks. In addition, except when you allocate memory and resize relocatable blocks, it can't move relocatable blocks around locked relocatable blocks (just as it can't move them around nonrelocatable blocks). Thus, locking a block in the middle of the heap for long periods of time can increase heap fragmentation.

Locking and unlocking blocks every time you want to prevent a block from moving can become troublesome. Fortunately, the Memory Manager moves unlocked, relocatable blocks only at well-defined, predictable times. In general, each routine description in *Inside Macintosh* indicates whether the routine could move or purge memory. If you do not call any of those routines in a section of code, you can rely on all blocks to remain stationary while that code executes. Note that the Segment Manager might move memory if you call a routine located in a segment that is not currently resident in memory. See "Loading Code Segments" on page 5-263 for details.

Purging and Reallocating Relocatable Blocks

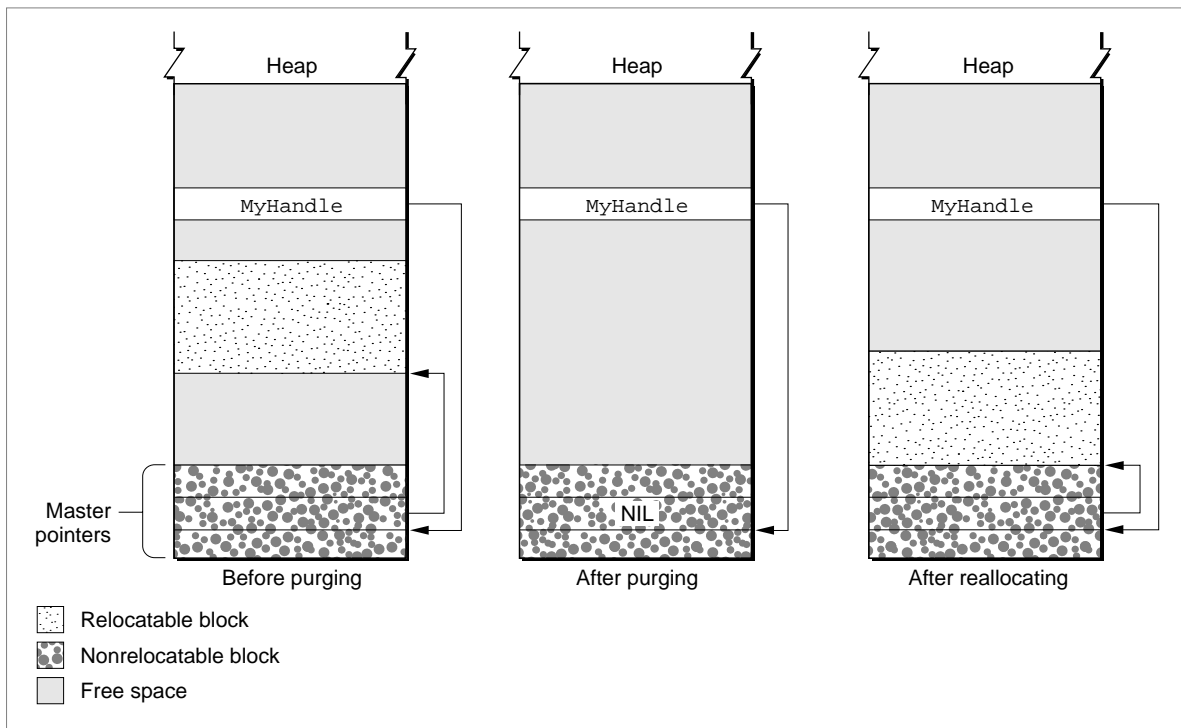
One advantage of relocatable blocks is that you can use them to store information that you would like to keep in memory to make your application more efficient, but that you don't really need if available memory space becomes low. For example, your application might, at the beginning of its execution, load user preferences from a preferences file into a relocatable block. As long as the block remains in memory, your application can access information from the preferences file without actually reopening the file. However, reopening the file probably wouldn't take enough time to justify keeping the block in memory if memory space were scarce.

By making a relocatable block **purgeable**, you allow the Memory Manager to free the space it occupies if necessary. If you later want to prohibit the Memory Manager from freeing the space occupied by a relocatable block, you can make the block **unpurgeable**. You can use the `HPurge` and `HNoPurge` procedures to change back and forth between these two states. A block you create by calling `NewHandle` is initially unpurgeable.

Once you make a relocatable block purgeable, you should subsequently check handles to that block before using them if you call any of the routines that could move or purge memory. If a handle's master pointer is set to `NIL`, then the Operating System has purged its block. To use the information formerly in the block, you must reallocate space for it (perhaps by calling the `ReallocateHandle` procedure) and then reconstruct its contents (for example, by rereading the preferences file).

Figure 5-10 illustrates the purging and reallocating of a relocatable block. When the block is purged, its master pointer is set to `NIL`. When it is reallocated, the handle correctly references a new block, but that block's contents are initially undefined.

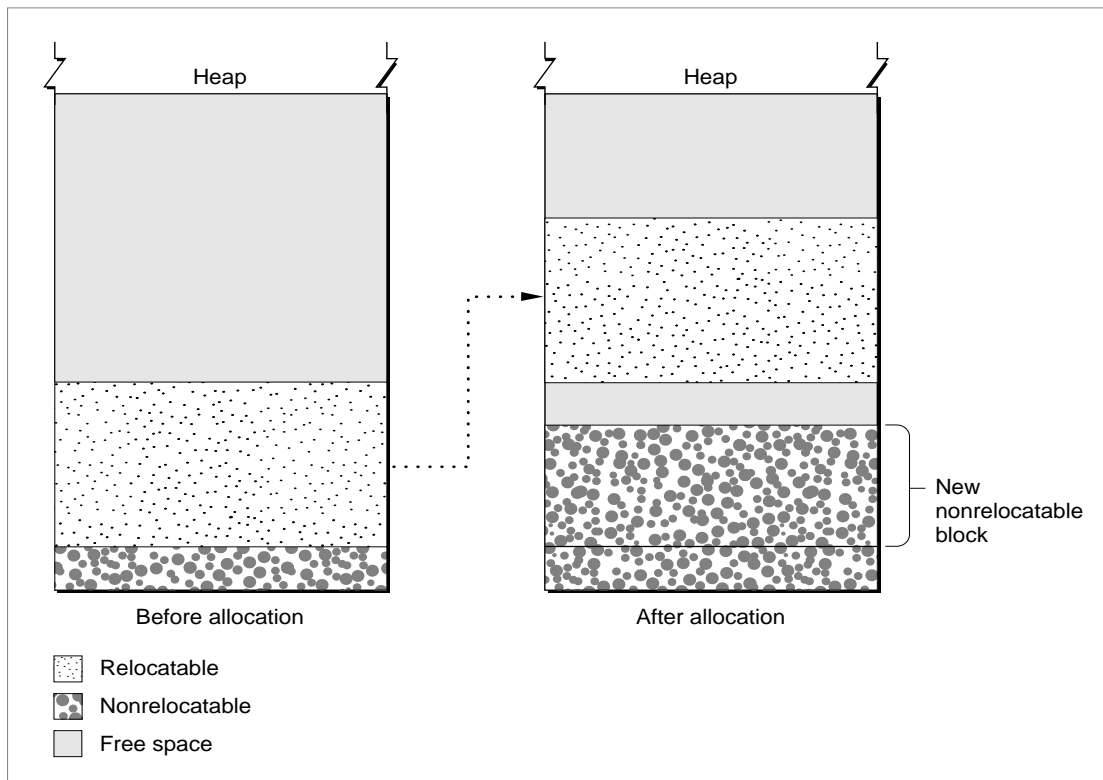
Figure 5-10 Purging and reallocating a relocatable block



Memory Reservation

The Memory Manager does its best to prevent situations in which nonrelocatable blocks in the middle of the heap trap relocatable blocks. When it allocates new nonrelocatable blocks, it attempts to **reserve** memory for them as low in the heap as possible. The Memory Manager reserves memory for a nonrelocatable block by moving unlocked relocatable blocks upward until it has created a space large enough for the new block. When the Memory Manager can successfully pack all nonrelocatable blocks into the bottom of the heap, no nonrelocatable block can trap a relocatable block, and it has successfully prevented heap fragmentation.

Figure 5-11 illustrates how the Memory Manager allocates nonrelocatable blocks. Although it could place a block of the requested size at the top of the heap, it instead reserves space for the block as close to the bottom of the heap as possible and then puts the block into that reserved space. During this process, the Memory Manager might even move a relocatable block over a nonrelocatable block to make room for another nonrelocatable block.

Figure 5-11 Allocating a nonrelocatable block

When allocating a new relocatable block, you can, if you want, manually reserve space for the block by calling the `ReserveMem` procedure. If you do not, the Memory Manager looks for space big enough for the block as low in the heap as possible, but it does not create space near the bottom of the heap for the block if there is already enough space higher in the heap.

Heap Purging and Compaction

When your application attempts to allocate memory (for example, by calling either the `NewPtr` or `NewHandle` function), the Memory Manager might need to **compact** or **purge** the heap to free memory and to fuse many small free blocks into fewer large free blocks. The Memory Manager first tries to obtain the requested amount of space by compacting the heap; if compaction fails to free the required amount of space, the Memory Manager then purges the heap.

When compacting the heap, the Memory Manager moves unlocked, relocatable blocks down until they reach nonrelocatable blocks or locked, relocatable blocks. You can compact the heap manually, by calling either the `CompactMem` function or the `MaxMem` function.

In a purge of the heap, the Memory Manager sequentially purges unlocked, purgeable relocatable blocks until it has freed enough memory or until it has purged all such blocks. It purges a block by deallocating it and setting its master pointer to `NIL`.

If you want, you can manually purge a few blocks or an entire heap in anticipation of a memory shortage. To purge an individual block manually, call the `EmptyHandle` procedure. To purge your entire heap manually, call the `PurgeMem` procedure or the `MaxMem` function.

Note

In general, you should let the Memory Manager purge and compact your heap, instead of performing these operations yourself. ♦

Heap Fragmentation

Heap fragmentation can slow your application by forcing the Memory Manager to compact or purge your heap to satisfy a memory-allocation request. In the worst cases, when your heap is severely fragmented by locked or nonrelocatable blocks, it might be impossible for the Memory Manager to find the requested amount of contiguous free space, even though that much space is actually free in your heap. This can have disastrous consequences for your application. For example, if the Memory Manager cannot find enough room to load a required code segment, your application will crash.

Obviously, it is best to minimize the amount of fragmentation that occurs in your application heap. It might be tempting to think that because the Memory Manager controls the movement of blocks in the heap, there is little that you can do to prevent heap fragmentation. In reality, however, fragmentation does not strike your application's heap by chance. Once you understand the major causes of heap fragmentation, you can follow a few simple rules to minimize it.

The primary causes of heap fragmentation are indiscriminate use of nonrelocatable blocks and indiscriminate locking of relocatable blocks. Each of these creates immovable blocks in your heap, thus creating “roadblocks” for the Memory Manager when it rearranges the heap to maximize the amount of contiguous free space. You can significantly reduce heap fragmentation simply by exercising care when you allocate nonrelocatable blocks and when you lock relocatable blocks.

Throughout this section, you should keep in mind the following rule: the Memory Manager can move a relocatable block around a nonrelocatable block (or a locked relocatable block) at these times only:

- When the Memory Manager reserves memory for a nonrelocatable block (or when you manually reserve memory before allocating a block), it can move unlocked, relocatable blocks upward over nonrelocatable blocks to make room for the new block as low in the heap as possible.
- When you attempt to resize a relocatable block, the Memory Manager can move that block around other blocks if necessary.

In contrast, the Memory Manager cannot move relocatable blocks over nonrelocatable blocks during compaction of the heap.

Deallocating Nonrelocatable Blocks

One of the most common causes of heap fragmentation is also one of the most difficult to avoid. The problem occurs when you dispose of a nonrelocatable block in the middle of the pile of nonrelocatable blocks at the bottom of the heap. Unless you immediately allocate another nonrelocatable block of the same size, you create a gap where the nonrelocatable block used to be. If you later allocate a slightly smaller, nonrelocatable block, that gap shrinks. However, small gaps are inefficient because of the small likelihood that future memory allocations will create blocks small enough to occupy the gaps.

It would not matter if the first block you allocated after deleting the nonrelocatable block were relocatable. The Memory Manager would place the block in the gap if possible. If you were later to allocate a nonrelocatable block as large as or smaller than the gap, the new block would take the place of the relocatable block, which would join other relocatable blocks in the middle of the heap, as desired. However, the new nonrelocatable block might be smaller than the original nonrelocatable block, leaving a small gap.

Whenever you dispose of a nonrelocatable block that you have allocated, you create small gaps, unless the next nonrelocatable block you allocate happens to be the same size as the disposed block. These small gaps can lead to heavy fragmentation over the course of your application's execution. Thus, you should try to avoid disposing of and then reallocating nonrelocatable blocks during program execution.

Reserving Memory

Another cause of heap fragmentation ironically occurs because of a limitation of memory reservation, a process designed to prevent it. Memory reservation never makes fragmentation worse than it would be if there were no memory reservation. Ordinarily, memory reservation ensures that allocating nonrelocatable blocks in the middle of your application's execution causes no problems. Occasionally, however, memory reservation can cause fragmentation, either when it succeeds but leaves small gaps in the reserved space, or when it fails and causes a nonrelocatable block to be allocated in the middle of the heap.

The Memory Manager uses memory reservation to create space for nonrelocatable blocks as low as possible in the heap. (You can also manually reserve memory for relocatable blocks, but you rarely need to do so.) However, when the Memory Manager moves a block up during memory reservation, that block cannot overlap its previous location. As a result, the Memory Manager might need to move the relocatable block up more than is necessary to contain the new nonrelocatable block, thereby creating a gap between the top of the new block and the bottom of the relocated block. (See Figure 5-11 on page 5-255.)

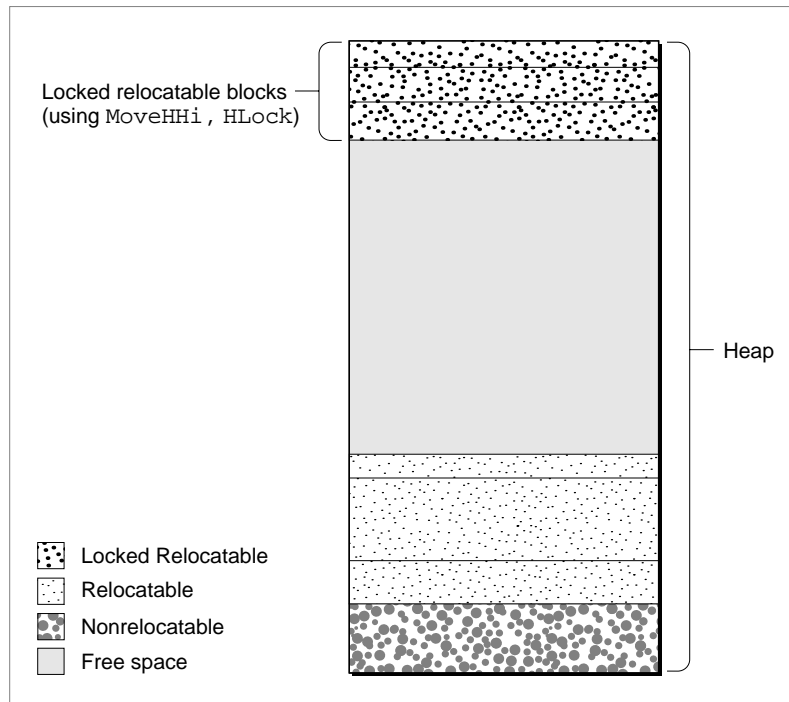
Memory reservation can also fragment the heap if there is not enough space in the heap to move the relocatable block up. In this case, the Memory Manager allocates the new nonrelocatable block above the relocatable block. The relocatable block cannot then move over the nonrelocatable block, except during the times described previously.

Locking Relocatable Blocks

Locked relocatable blocks present a special problem. When relocatable blocks are locked, they can cause as much heap fragmentation as nonrelocatable blocks. One solution is to reserve memory for all relocatable blocks that might at some point need to be locked, and to leave them locked for as long as they are allocated. This solution has drawbacks, however, because then the blocks would lose any flexibility that being relocatable otherwise gives them. Deleting a locked relocatable block can create a gap, just as deleting a nonrelocatable block can.

An alternative partial solution is to move relocatable blocks to the top of the heap before locking them. The `MoveHHi` procedure allows you to move a relocatable block upward until it reaches the top of the heap, a nonrelocatable block, or a locked relocatable block. This has the effect of partitioning the heap into four areas, as illustrated in Figure 5-12. At the bottom of the heap are the nonrelocatable blocks. Above those blocks are the unlocked relocatable blocks. At the top of the heap are locked relocatable blocks. Between the locked relocatable blocks and the unlocked relocatable blocks is an area of free space. The principal idea behind moving relocatable blocks to the top of the heap and locking them there is to keep the contiguous free space as large as possible.

Figure 5-12 An effectively partitioned heap



Using `MoveHHi` is, however, not always a perfect solution to handling relocatable blocks that need to be locked. The `MoveHHi` procedure moves a block upward only until it reaches either a nonrelocatable block or a locked relocatable block. Unlike `NewPtr` and `ReserveMem`, `MoveHHi` does not currently move a relocatable block around one that is not relocatable.

Even if `MoveHHi` succeeds in moving a block to the top area of the heap, unlocking or deleting locked blocks can cause fragmentation if you don't unlock or delete those blocks beginning with the lowest locked block. A relocatable block that is locked at the top area of the heap for a long period of time could trap other relocatable blocks that were locked for short periods of time but then unlocked.

This suggests that you need to treat relocatable blocks locked for a long period of time differently from those locked for a short period of time. If you plan to lock a relocatable block for a long period of time, you should reserve memory for it at the bottom of the heap before allocating it, then lock it for the duration of your application's execution (or as long as the block remains allocated). Do not reserve memory for relocatable blocks you plan to allocate for only short periods of time. Instead, move them to the top of the heap (by calling `MoveHHi`) and then lock them.

Note

You should call `MoveHHi` only on blocks located in your application heap. Don't call `MoveHHi` on relocatable blocks in the system heap. Desk accessories should not call `MoveHHi`. ♦

In practice, you apply the same rules to relocatable blocks that you reserve space for and leave permanently locked as you apply to nonrelocatable blocks: Try not to allocate such blocks in the middle of your application's execution, and don't dispose of and reallocate such blocks in the middle of your application's execution.

After you lock relocatable blocks temporarily, you don't need to move them manually back into the middle area when you unlock them. Whenever the Memory Manager compacts the heap or moves another relocatable block to the top heap area, it brings all unlocked relocatable blocks at the bottom of that partition back into the middle area. When moving a block to the top area, be sure to call `MoveHHi` on the block and then lock the block, in that order.

Allocating Nonrelocatable Blocks

As you have seen, there are two reasons for not allocating nonrelocatable blocks during the middle of your application's execution. First, if you also dispose of nonrelocatable blocks in the middle of your application's execution, then allocation of new nonrelocatable blocks is likely to create small gaps, as discussed earlier. Second, even if you never dispose of nonrelocatable blocks until your application terminates, memory reservation is an imperfect process, and the Memory Manager could occasionally place new nonrelocatable blocks above relocatable blocks.

There is, however, an exception to the rule that you should not allocate nonrelocatable blocks in the middle of your application's execution. Sometimes you need to allocate a nonrelocatable block only temporarily. If between the times that you allocate and dispose of a nonrelocatable block, you allocate no additional nonrelocatable blocks and do not attempt to compact the heap, then you have done no harm. The temporary block cannot create a new gap because the Memory Manager places no other block over the temporary block.

Summary of Preventing Fragmentation

Avoiding heap fragmentation is not difficult. It simply requires that you follow a few rules as closely as possible. Remember that allocation of even a small nonrelocatable block in the middle of your heap can ruin a scheme to prevent fragmentation of the heap, because the Memory Manager does not move relocatable blocks around nonrelocatable blocks when you call `MoveHHI` or when it attempts to compact the heap.

If you adhere to the following rules, you are likely to avoid significant heap fragmentation:

- At the beginning of your application's execution, call the `MaxApplZone` procedure once and the `MoreMasters` procedure enough times so that the Memory Manager never needs to call it for you.
- Try to anticipate the maximum number of nonrelocatable blocks you will need and allocate them at the beginning of your application's execution.
- Avoid disposing of and then reallocating nonrelocatable blocks during your application's execution.
- When allocating relocatable blocks that you need to lock for long periods of time, use the `ReserveMem` procedure to reserve memory for them as close to the bottom of the heap as possible, and lock the blocks immediately after allocating them.
- If you plan to lock a relocatable block for a short period of time and allocate nonrelocatable blocks while it is locked, use the `MoveHHI` procedure to move the block to the top of the heap and then lock it. When the block no longer needs to be locked, unlock it.
- Remember that you need to lock a relocatable block only if you call a routine that could move or purge memory and you then use a dereferenced handle to the relocatable block, or if you want to use a dereferenced handle to the relocatable block at interrupt time.

Perhaps the most difficult restriction is to avoid disposing of and then reallocating nonrelocatable blocks in the middle of your application's execution. Some Toolbox routines require you to use nonrelocatable blocks, and it is not always easy to anticipate how many such blocks you will need. If you must allocate and dispose of blocks in the middle of your program's execution, you might want to place used blocks into a linked list of free blocks instead of disposing of them. If you know how many nonrelocatable blocks of a certain size your application is likely to need, you can add that many to the beginning of the list at the beginning of your application's execution. If you need a nonrelocatable block later, you can check the linked list for a block of the exact size instead of simply calling the `NewPtr` function.

Dangling Pointers

Accessing a relocatable block by double indirection, through its handle instead of through its master pointer, requires an extra memory reference. For efficiency, you might sometimes want to **dereference** the handle—that is, make a copy of the block's master pointer—and then use that pointer to access the block by single indirection. When you do this, however, you need to be particularly careful. Any operation that allocates space from the heap might cause the relocatable block to be moved or purged. In that event, the block's master pointer is correctly updated, but your copy of the master pointer is not. As a result, your copy of the master pointer is a **dangling pointer**.

Dangling pointers are likely to make your application crash or produce garbled output. Unfortunately, it is often easy during debugging to overlook situations that could leave pointers dangling, because pointers dangle only if the relocatable blocks that they reference actually move. Routines that can move or purge memory do not necessarily do so unless memory space is tight. Thus, if you improperly dereference a handle in a section of code, that code might still work properly most of the time. If, however, a dangling pointer does cause errors, they can be very difficult to trace.

This section describes a number of situations that can cause dangling pointers and suggests some ways to avoid them.

Compiler Dereferencing

Some of the most difficult dangling pointers to isolate are not caused by any explicit dereferencing on your part, but by implicit dereferencing on the part of the compiler. For example, suppose you use a handle called `myHandle` to access the fields of a record in a relocatable block. You might use Pascal's `WITH` statement to do so, as follows:

```
WITH myHandle^^ DO
  BEGIN
    . . .
  END;
```

A compiler is likely to dereference `myHandle` so that it can access the fields of the record without double indirection. However, if the code between the `BEGIN` and `END` statements causes the Memory Manager to move or purge memory, you are likely to end up with a dangling pointer.

The easiest way to prevent dangling pointers is simply to lock the relocatable block whose data you want to read or write. Because the block is locked and cannot move,

the master pointer is guaranteed always to point to the beginning of the block's data. Listing 5-1 illustrates one way to avoid dangling pointers by locking a relocatable block.

Listing 5-1 Locking a block to avoid dangling pointers

```
VAR
    origState: SignedByte;    {original attributes of handle}

origState := HGetState(Handle(myData)); {get handle attributes}
MoveHHi(Handle(myData));           {move the handle high}
HLock(Handle(myData));             {lock the handle}
WITH myData^^ DO                   {fill in window data}
    BEGIN
        editRec := TENew(gDestRect, gViewRect);
        vScroll := GetNewControl(rVScroll, myWindow);
        hScroll := GetNewControl(rHScroll, myWindow);
        fileRefNum := 0;
        windowDirty := FALSE;
    END;
HSetState(origState);              {reset handle attributes}
```

The handle `myData` needs to be locked before the `WITH` statement because the functions `TENew` and `GetNewControl` allocate memory and hence might move the block whose handle is `myData`.

You should be careful to lock blocks only when necessary, because locked relocatable blocks can increase heap fragmentation and slow down your application unnecessarily. You should lock a handle only if you dereference it, directly or indirectly, and then use a copy of the original master pointer after calling a routine that could move or purge memory. When you no longer need to reference the block with the master pointer, you should unlock the handle. In Listing 5-1, the handle `myData` is never explicitly unlocked. Instead, the original attributes of the handle are saved by calling `HGetState` and later are restored by calling `HSetState`. This strategy is preferable to just calling `HLock` and `HUnlock`.

A compiler can generate hidden dereferencing, and hence potential dangling pointers, in other ways, for instance, by assigning the result of a function that might move or purge blocks to a field in a record referenced by a handle. Such problems are particularly common in code that manipulates linked data structures. For example, you might use this code to allocate a new element of a linked list:

```
myHandle^^.nextHandle := NewHandle(sizeof(myLinkedElement));
```

This can cause problems because your compiler could dereference `myHandle` before calling `NewHandle`. Therefore, you should either lock `myHandle` before performing the allocation, or use a temporary variable to allocate the new handle, as in the following code:

```
tempHandle := NewHandle(sizeof(myLinkedElement));
myHandle^.nextHandle := tempHandle;
```

Passing fields of records as arguments to routines that might move or purge memory can cause similar problems, if the records are in relocatable blocks referred to with handles. Problems arise only when you pass a field by reference rather than by value. Pascal conventions call for all arguments larger than 4 bytes to be passed by reference. In Pascal, a variable is also passed by reference when the routine called requests a variable parameter. Both of the following lines of code could leave a pointer dangling:

```
TEUpdate(hTE^.viewRect, hTE);
InvalRect(theControl^.ctrlRect);
```

These problems occur because a compiler may dereference a handle before calling the routine to which you pass the handle. Then, that routine may move memory before it uses the dereferenced handle, which might then be invalid. As before, you can solve these problems by locking the handles or using temporary variables.

Loading Code Segments

If you call an application-defined routine located in a code segment that is not currently in RAM, the Segment Manager might need to move memory when loading that code segment, thus jeopardizing any dereferenced handles you might be using. For example, suppose you call an application-defined procedure `ManipulateData`, which manipulates some data at an address passed to it in a variable parameter.

```
PROCEDURE MyRoutine;
BEGIN
    ...
    ManipulateData(myHandle^);
    ...
END;
```

You can create a dangling pointer if `ManipulateData` and `MyRoutine` are in different segments, and the segment containing `ManipulateData` is not loaded when `MyRoutine` is executed. You can do this because you've passed a dereferenced copy of `myHandle` as an argument to `ManipulateData`. If the Segment Manager must allocate a new relocatable block for the segment containing `ManipulateData`, it might move `myHandle` to do so. If so, the dereferenced handle would dangle. A similar problem can occur if you assign the result of a function in a nonresident code segment to a field in a record referred to by a handle.

You need to be careful even when passing a field in a record referenced by a handle to a routine in the same code segment as the caller, or when assigning the result of a function in the same code segment to such a field. If that routine could call a Toolbox routine that might move or purge memory, or call a routine in a different, nonresident code segment, then you could indirectly cause a pointer to dangle.

Callback Routines

Code segmentation can also lead to a different type of dangling-pointer problem when you use callback routines. The problem rarely arises, but it is difficult to debug. Some Toolbox routines require that you pass a pointer to a procedure in a variable of type `ProcPtr`. Ordinarily, it does not matter whether the procedure you pass in such a variable is in the same code segment as the routine that calls it or in a different code segment. For example, suppose you call `TrackControl` as follows:

```
myPart := TrackControl(myControl, myEvent.where, @MyCallback);
```

If `MyCallback` were in the same code segment as this line of code, then a compiler would pass to `TrackControl` the absolute address of the `MyCallback` procedure. If it were in a different code segment, then the compiler would take the address from the jump table entry for `MyCallback`. Either way, `TrackControl` should call `MyCallback` correctly.

Occasionally, you might use a variable of type `ProcPtr` to hold the address of a callback procedure and then pass that address to a routine. Here is an example:

```
myProc := @MyCallback;
...
myPart := TrackControl(myControl, myEvent.where, myProc);
```

As long as these lines of code are in the same code segment and the segment is not unloaded between the execution of those lines, the preceding code should work perfectly. Suppose, however, that `myProc` is a global variable, and the first line of the code is in a different segment from the call to `TrackControl`. Suppose, further, that the `MyCallback` procedure is in the same segment as the first line of the code (which is in a different segment from the call to `TrackControl`). Then, the compiler might place the absolute address of the `MyCallback` routine into the variable `myProc`. The compiler cannot realize that you plan to use the variable in a different code segment from the one that holds both the routine you are referencing and the routine you are using to initialize the `myProc` variable. Because `MyCallback` and the call to `TrackControl` are in different code segments, the `TrackControl` procedure requires that you pass an address in the jump table, not an absolute address. Thus, in this hypothetical situation, `myProc` would reference `MyCallback` incorrectly.

To avoid this problem, make sure to place in the same segment any code in which you assign a value to a variable of type `ProcPtr` and any code in which you use that

variable. If you must put them in different code segments, then be sure that you place the callback routine in a code segment different from the one that initializes the variable.

Note

Some development systems allow you to specify compiler options that force jump table references to be generated for routine addresses. If you specify those options, the problems described in this section cannot arise. ♦

Invalid Handles

An invalid handle refers to the wrong area of memory, just as a dangling pointer does. There are three types of invalid handles: empty handles, disposed handles, and fake handles. You must avoid empty, disposed, or fake handles as carefully as dangling pointers. Fortunately, it is generally easier to detect, and thus to avoid, invalid handles.

Disposed Handles

A **disposed handle** is a handle whose associated relocatable block has been disposed of. When you dispose of a relocatable block (perhaps by calling the procedure `DisposeHandle`), the Memory Manager does not change the value of any handle variables that previously referenced that block. Instead, those variables still hold the address of what once was the relocatable block's master pointer. Because the block has been disposed of, however, the contents of the master pointer are no longer defined. (The master pointer might belong to a subsequently allocated relocatable block, or it could become part of a linked list of unused master pointers maintained by the Memory Manager.)

If you accidentally use a handle to a block you have already disposed of, you can obtain unexpected results. In the best cases, your application will crash. In the worst cases, you will get garbled data. It might, however, be difficult to trace the cause of the garbled data, because your application can continue to run for quite a while before the problem begins to manifest itself.

You can avoid these problems quite easily by assigning the value `NIL` to the handle variable after you dispose of its associated block. By doing so, you indicate that the handle does not point anywhere in particular. If you subsequently attempt to operate on such a block, the Memory Manager will probably generate a `nilHandleErr` result code. If you want to make certain that a handle is not disposed of before operating on a relocatable block, you can test whether the value of the handle is `NIL`, as follows:

```
IF myHandle <> NIL THEN
    ...;                {handle is valid, so we can operate on it here}
```

Note

This test is useful only if you manually assign the value `NIL` to all disposed handles. The Memory Manager does not do that automatically. ♦

Empty Handles

An **empty handle** is a handle whose master pointer has the value `NIL`. When the Memory Manager purges a relocatable block, for example, it sets the block's master pointer to `NIL`. The space occupied by the master pointer itself remains allocated, and handles to the purged block continue to point to the master pointer. This is useful, because if you later reallocate space for the block by calling `ReallocateHandle`, the master pointer will be updated and all existing handles will correctly access the reallocated block.

Note

Don't confuse empty handles with **0-length handles**, which are handles whose associated block has a size of 0 bytes. A 0-length handle has a non-`NIL` master pointer and a block header. ♦

Once again, however, inadvertently using an empty handle can give unexpected results or lead to a system crash. In the Macintosh Operating System, `NIL` technically refers to memory location 0. But this memory location holds a value. If you doubly dereference an empty handle, you reference whatever data is found at that location, and you could obtain unexpected results that are difficult to trace.

You can check for empty handles much as you check for disposed handles. Assuming you set handles to `NIL` when you dispose of them, you can use the following code to determine whether a handle both points to a valid master pointer and references a nonempty relocatable block:

```
IF myHandle <> NIL THEN
    IF myHandle^ <> NIL THEN
        ...           {we can operate on the relocatable block here}
```

Note that because Pascal evaluates expressions completely, you need two `IF-THEN` statements rather than one compound statement in case the value of the handle itself is `NIL`. Most compilers, however, allow you to use "short-circuit" Boolean operators to minimize the evaluation of expressions. For example, if your compiler uses the operator `&` as a short-circuit operator for `AND`, you could rewrite the preceding code like this:

```
IF (myHandle <> NIL) & (myHandle^ <> NIL) THEN
    ...           {we can operate on the relocatable block here}
```

In this case, the second expression is evaluated only if the first expression evaluates to `TRUE`.

Note

The availability and syntax of short-circuit Boolean operators are compiler dependent. Check the documentation for your development system to see whether you can use such operators. ♦

It is useful during debugging to set memory location 0 to an odd number, such as \$50FFC001. This causes the Operating System to crash immediately if you attempt to dereference an empty handle. This is useful, because you can immediately fix problems that might otherwise require extensive debugging.

Fake Handles

A **fake handle** is a handle that was not created by the Memory Manager. Normally, you create handles by either directly or indirectly calling the Memory Manager function `NewHandle` (or one of its variants, such as `NewHandleClear`). You create a fake handle—usually inadvertently—by directly assigning a value to a variable of type `Handle`, as illustrated in Listing 5-2.

Listing 5-2 Creating a fake handle

```
FUNCTION MakeFakeHandle: Handle;      {DON'T USE THIS FUNCTION!}
CONST
    kMemoryLoc = $100;                {a random memory location}
VAR
    myHandle:   Handle;
    myPointer:  Ptr;
BEGIN
    myPointer := Ptr(kMemoryLoc);      {the address of some memory}
    myHandle := @myPointer;           {the address of a pointer}
    MakeFakeHandle := myHandle;
END;
```

▲ WARNING

The technique for creating a fake handle shown in Listing 5-2 is included for illustrative purposes only. Your application should never create fake handles. ▲

Remember that a real handle contains the address of a master pointer. The fake handle manufactured by the function `MakeFakeHandle` in Listing 5-2 contains an address that may or may not be the address of a master pointer. If it isn't the address of a master pointer, then you virtually guarantee chaotic results if you pass the fake handle to a system software routine that expects a real handle.

For example, suppose you pass a fake handle to the `MoveHHI` procedure. After allocating a new relocatable block high in the heap, `MoveHHI` is likely to copy the data from the original block to the new block by dereferencing the handle and using, supposedly, a master pointer. Because, however, the value of a fake handle probably isn't the address of a master pointer, `MoveHHI` copies invalid data. (Actually, it's unlikely that `MoveHHI` would ever get that far; probably it would run into problems when attempting to determine the size of the original block from the block header.)

Not all fake handles are as easy to spot as those created by the `MakeFakeHandle` function defined in Listing 5-2. You might, for instance, attempt to copy the data in an existing record (`myRecord`) into a new handle, as follows:

```
myHandle := NewHandle(SizeOf(myRecord)); {create a new handle}
myHandle^ := @myRecord;                 {DON'T DO THIS!}
```

The second line of code does *not* make `myHandle` a handle to the beginning of the `myRecord` record. Instead, it overwrites the master pointer with the address of that record, making `myHandle` a fake handle.

▲ WARNING

Never assign a value directly to a master pointer. ▲

A correct way to create a new handle to some existing data is to make a copy of the data using the `PtrToHand` function, as follows:

```
myErr := PtrToHand(@myRecord, myHandle, SizeOf(myRecord));
```

The Memory Manager provides a set of pointer- and handle-manipulation routines that can help you avoid creating fake handles. See the chapter “Memory Manager” in this book for details on those routines.

Low-Memory Conditions

It is particularly important to make sure that the amount of free space in your application heap never gets too low. For example, you should never deplete the available heap memory to the point that it becomes impossible to load required code segments. As you have seen, your application will crash if the Segment Manager is called to load a required code segment and there is not enough contiguous free memory to allocate a block of the appropriate size.

You can take several steps to help maximize the amount of free space in your heap. For example, you can mark as purgeable any relocatable blocks whose contents could easily be reconstructed. By making a block purgeable, you give the Memory Manager the freedom to release that space if heap memory becomes low. You can also help maximize the available heap memory by intelligently segmenting your application’s executable code and by periodically unloading any unneeded segments. The standard way to do this is to unload every nonessential segment at the end of your application’s main event loop. (See the chapter “Segment Manager” in *Inside Macintosh: Processes* for a complete discussion of code-segmentation techniques.)

Memory Cushions

These two measures—making blocks purgeable and unloading segments—help you only by releasing blocks that have already been allocated. It is even more important to make sure, *before* you attempt to allocate memory directly, that you don’t deplete the available heap memory. Before you call `NewHandle` or `NewPtr`, you should check that, if the requested amount of memory were in fact allocated, the remaining amount of space

free in the heap would not fall below a certain threshold. The free memory defined by that threshold is your **memory cushion**. You should not simply inspect the handle or pointer returned to you and make sure that its value isn't `NIL`, because you might have succeeded in allocating the space you requested but left the amount of free space dangerously low.

You also need to make sure that indirect memory allocation doesn't cut into the memory cushion. When, for example, you call `GetNewDialog`, the Dialog Manager might need to allocate space for a dialog record; it also needs to allocate heap space for the dialog item list and any other custom items in the dialog. Before calling `GetNewDialog`, therefore, you need to make sure that the amount of space left free after the call is greater than your memory cushion.

The execution of some system software routines requires significant amounts of memory in your heap. For example, some QuickDraw operations on regions can temporarily allocate fairly large amounts of space in your heap. Some of these system software routines, however, do little or no checking to see that your heap contains the required amount of free space. They either assume that they will get whatever memory they need or they simply issue a system error when they don't get the needed memory. In either case, the result is usually a system crash.

You can avoid these problems by making sure that there is always enough space in your heap to handle these hidden memory allocations. Experience has shown that 40 KB is a reasonably safe size for this memory cushion. If you can consistently maintain that amount of space free in your heap, you can be reasonably certain that system software routines will get the memory they need to operate. You also generally need a larger cushion (about 70 KB) when printing.

Memory Reserves

Unfortunately, there are times when you might need to use some of the memory in the cushion yourself. It is better, for instance, to dip into the memory cushion, if necessary, to save a user's document than to reject the request to save the document. Some actions your application performs should not be rejectable simply because they require it to reduce the amount of free space below a desired minimum.

Instead of relying on just the free memory of a memory cushion, you can allocate a **memory reserve**, some additional emergency storage that you release when free memory becomes low. The important difference between this memory reserve and the memory cushion is that the memory reserve is a block of allocated memory, which you release whenever you detect that essential tasks have dipped into the memory cushion.

That emergency memory reserve might provide enough memory to compensate for any essential tasks that you fail to anticipate. Because you allow essential tasks to dip into the memory cushion, the release itself of the memory reserve should not be a cause for alarm. Using this scheme, your application releases the memory reserve as a precautionary measure during ordinary operation. Ideally, however, the application should never actually deplete the memory cushion and use the memory reserve.

Grow-Zone Functions

The Memory Manager provides a particularly easy way for you to make sure that the emergency memory reserve is released when necessary. You can define a **grow-zone function** that is associated with your application heap. The Memory Manager calls your heap's grow-zone function only after other techniques of freeing memory to satisfy a memory request fail (that is, after compacting and purging the heap and extending the heap zone to its maximum size). The grow-zone function can then take appropriate steps to free additional memory.

A grow-zone function might dispose of some blocks or make some unpurgeable blocks purgeable. When the function returns, the Memory Manager once again purges and compacts the heap and tries to reallocate memory. If there is still insufficient memory, the Memory Manager calls the grow-zone function again (but only if the function returned a nonzero value the previous time it was called). This mechanism allows your grow-zone function to release just a little bit of memory at a time. If the amount it releases at any time is not enough, the Memory Manager calls it again and gives it the opportunity to take more drastic measures. As the most drastic step to freeing memory in your heap, you can release the emergency reserve.

Using Memory

This section describes how you can use the Memory Manager to perform the most typical memory management tasks. In particular, this section shows how you can

- set up your application heap at application launch time
- determine how much free space is available in your application heap
- allocate and release blocks of memory in your heap
- define and install a grow-zone function

The techniques described in this section are designed to minimize fragmentation of your application heap and to ensure that your application always has sufficient memory to complete any essential operations. Many of these techniques incorporate the heap memory cushion and emergency memory reserve discussed in “Low-Memory Conditions,” beginning on page 5-268.

Note

This section describes relatively simple memory-management techniques. Depending on the requirements of your application, you might want to manage your heap memory differently. ♦

Setting Up the Application Heap

When the Process Manager launches your application, it calls the Memory Manager to create and initialize a memory partition for your application. The Process Manager then

loads code segments into memory and sets up the stack, heap, and A5 world (including the jump table) for your application.

To help prevent heap fragmentation, you should also perform some setup of your own early in your application's execution. Depending on the needs of your application, you might want to

- change the size of your application's stack
- expand the heap to the heap limit
- allocate additional master pointer blocks

The following sections describe in detail how and when to perform these operations.

Changing the Size of the Stack

Most applications allocate space on their stack in a predictable way and do not need to monitor stack space during their execution. For these applications, stack usage usually reaches a maximum in some heavily nested routine. If the stack in your application can never grow beyond a certain size, then to avoid collisions between your stack and heap you simply need to ensure that your stack is large enough to accommodate that size.

If you never encounter system error 28 (generated by the stack sniffer when it detects a collision between the stack and the heap) during application testing, then you probably do not need to increase the size of your stack.

Some applications, however, rely heavily on recursive programming techniques, in which one routine repeatedly calls itself or a small group of routines repeatedly call each other. In these applications, even routines with just a few local variables can cause stack overflow, because each time a routine calls itself, a new copy of that routine's parameters and variables is appended to the stack. The problem can become particularly acute if one or more of the local variables is a string, which can require up to 256 bytes of stack space.

You can help prevent your application from crashing because of insufficient stack space by expanding the size of your stack. If your application does not depend on recursion, you should do this only if you encounter system error 28 during testing. If your application does depend on recursion, you might consider expanding the stack so that your application can perform deeply nested recursive computations. In addition, some object-oriented languages (for example, C++) allocate space for objects on the stack. If you are using one of these languages, you might need to expand your stack.

Note

If you are programming in LISP or another language that depends extensively on recursion, your development system might allocate memory for local variables in the heap rather than on the stack. If so, expanding the size of the stack is not helpful. Consult your development system's documentation for details on how it allocates memory. ♦

To increase the size of your stack, you simply reduce the size of your heap. Because the heap cannot grow above the boundary contained in the `ApplLimit` global variable, you can lower the value of `ApplLimit` to limit the heap's growth. By lowering `ApplLimit`,

technically you are not making the stack bigger; you are just preventing collisions between it and the heap.

By default, the stack can grow to 8 KB on Macintosh computers without Color QuickDraw and to 32 KB on computers with Color QuickDraw. (The size of the stack for a faceless background process is always 8 KB, whether Color QuickDraw is present or not.) You should never decrease the size of the stack, because future versions of system software might increase the default amount of space allocated for the stack. For the same reason, you should not set the stack to a predetermined absolute size or calculate a new absolute size for the stack based on the microprocessor's type. If you must modify the size of the stack, you should increase the stack size only by some relative amount that is sufficient to meet the increased stack requirements of your application. There is no maximum size to which the stack can grow.

Listing 5-3 defines a procedure that increases the stack size by a given value. It does so by determining the current heap limit, subtracting the value of the `extraBytes` parameter from that value, and then setting the application limit to the difference.

Listing 5-3 Increasing the amount of space allocated for the stack

```
PROCEDURE IncreaseStackSize (extraBytes: Size);
BEGIN
    SetApplLimit(Ptr(ORD4(GetApplLimit) - extraBytes));
END;
```

You should call this procedure at the beginning of your application, before you call the `MaxApplZone` procedure (as described in the next section). If you call `IncreaseStackSize` after you call `MaxApplZone`, it has no effect, because the `SetApplLimit` procedure cannot change the `ApplLimit` global variable to a value lower than the current top of the heap.

Note

Some compilers add to the beginning of your application some default initialization code that automatically calls `MaxApplZone`. You might need to specify a compiler directive that turns off such default initialization if you want to increase the size of the stack. Consult your development system's documentation for details. ♦

Expanding the Heap

Near the beginning of your application's execution, before you allocate any memory, you should call the `MaxApplZone` procedure to expand the application heap immediately to the application heap limit. If you do not do this, the Memory Manager gradually expands your heap as memory needs require. This gradual expansion can result in significant heap fragmentation if you have previously moved relocatable blocks to the top of the heap (by calling `MoveHHI`) and locked them (by calling `HLock`). When the heap grows beyond those locked blocks, they are no longer at the top of the heap. Your heap then remains fragmented for as long as those blocks remain locked.

Another advantage to calling `MaxApplZone` is that doing so is likely to reduce the number of relocatable blocks that are purged by the Memory Manager. The Memory Manager expands your heap to fulfill a memory request only after it has exhausted other methods of obtaining the required amount of space, including compacting the heap and purging blocks marked as purgeable. By expanding the heap to its limit, you can prevent the Memory Manager from purging blocks that it otherwise would purge. This, together with the fact that your heap is expanded only once, can make memory allocation significantly faster.

Note

As indicated in the previous section, you should call `MaxApplZone` only after you have expanded the stack, if necessary. ♦

Allocating Master Pointer Blocks

After calling `MaxApplZone`, you should call the `MoreMasters` procedure to allocate as many new nonrelocatable blocks of master pointers as your application is likely to need during its execution. Each block of master pointers in your application heap contains 64 master pointers. The Operating System allocates one block of master pointers as your application is loaded into memory, and every relocatable block you allocate needs one master pointer to reference it.

If, when you allocate a relocatable block, there are no unused master pointers in your application heap, the Memory Manager automatically allocates a new block of master pointers. For several reasons, however, you should try to prevent the Memory Manager from calling `MoreMasters` for you. First, `MoreMasters` executes more slowly if it has to move relocatable blocks up in the heap to make room for the new nonrelocatable block of master pointers. When your application first starts running, there are no such blocks that might have to be moved. Second, the new nonrelocatable block of master pointers is likely to fragment your application heap. At any time the Memory Manager is forced to call `MoreMasters` for you, there are already at least 64 relocatable blocks allocated in your heap. Unless all or most of those blocks are locked high in the heap (an unlikely situation), the new nonrelocatable block of master pointers might be allocated above existing relocatable blocks. This increases heap fragmentation.

To prevent this fragmentation, you should call `MoreMasters` at the beginning of your application enough times to ensure that the Memory Manager never needs to call it for you. For example, if your application never allocates more than 300 relocatable blocks in its heap, then five calls to the `MoreMasters` should be enough. It's better to call `MoreMasters` too many times than too few, so if your application usually allocates about 100 relocatable blocks but sometimes might allocate 1000 in a particularly busy session, you should call `MoreMasters` enough times at the beginning of the program to cover the larger figure.

You can determine empirically how many times to call `MoreMasters` by using a low-level debugger. First, remove all the calls to `MoreMasters` from your code and then give your application a rigorous workout, opening and closing windows, dialog boxes, and desk accessories as much as any user would. Then, find out from your debugger how many times the system called `MoreMasters`. To do so, count the nonrelocatable blocks

of size \$100 bytes (decimal 256, or 64×4). Because of Memory Manager size corrections, you should also count any nonrelocatable blocks of size \$108, \$10C, or \$110 bytes. (You should also check to make sure that your application doesn't allocate other nonrelocatable blocks of those sizes. If it does, subtract the number it allocates from the total.) Finally, call `MoreMasters` at least that many times at the beginning of your application.

Listing 5-4 illustrates a typical sequence of steps to configure your application heap and stack. The `DoSetUpHeap` procedure defined there increases the size of the stack by 32 KB, expands the application heap to its new limit, and allocates five additional blocks of master pointers.

Listing 5-4 Setting up your application heap and stack

```
PROCEDURE DoSetUpHeap;
CONST
    kExtraStackSize = $8000;           { 32 KB}
    kMoreMasterCalls = 5;              {for 320 master ptrs}
VAR
    count: Integer;
BEGIN
    IncreaseStackSize(kExtraStackSize); {increase stack size}
    MaxApplZone;                       {extend heap to limit}
    FOR count := 1 TO kMoreMasterCalls DO
        MoreMasters;                   {64 more master ptrs}
    END;
```

To reduce heap fragmentation, you should call `DoSetUpHeap` in a code segment that you never unload (possibly the main segment) rather than in a special initialization code segment. This is because `MoreMasters` allocates a nonrelocatable block. If you call `MoreMasters` from a code segment that is later purged, the new master pointer block is located above the purged space, thereby increasing fragmentation.

Determining the Amount of Free Memory

Because space in your heap is limited, you cannot usually honor every user request that would require your application to allocate memory. For example, every time the user opens a new window, you probably need to allocate a new window record and other associated data structures. If you allow the user to open windows endlessly, you risk running out of memory. This might adversely affect your application's ability to perform important operations such as saving existing data in a window.

It is important, therefore, to implement some scheme that prevents your application from using too much of its own heap. One way to do this is to maintain a memory cushion that can be used only to satisfy essential memory requests. Before allocating memory for any nonessential task, you need to ensure that the amount of memory that remains free after

the allocation exceeds the size of your memory cushion. You can do this by calling the function `IsMemoryAvailable` defined in Listing 5-5.

Listing 5-5 Determining whether allocating memory would deplete the memory cushion

```
FUNCTION IsMemoryAvailable (memRequest: LongInt): Boolean;
VAR
    total:    LongInt;    {total free memory if heap purged}
    contig:   LongInt;    {largest contiguous block if heap purged}
BEGIN
    PurgeSpace(total, contig);
    IsMemoryAvailable := ((memRequest + kMemCushion) < contig);
END;
```

The `IsMemoryAvailable` function calls the Memory Manager's `PurgeSpace` procedure to determine the size of the largest contiguous block that would be available if the application heap were purged; that size is returned in the `contig` parameter. If the size of the potential memory request together with the size of the memory cushion is less than the value returned in `contig`, `IsMemoryAvailable` is set to `TRUE`, indicating that it is safe to allocate the specified amount of memory; otherwise, `IsMemoryAvailable` returns `FALSE`.

Notice that the `IsMemoryAvailable` function does not itself cause the heap to be purged or compacted; the Memory Manager does so automatically when you actually attempt to allocate the memory.

Usually, the easiest way to determine how big to make your application's memory cushion is to experiment with various values. You should attempt to find the lowest value that allows your application to execute successfully no matter how hard you try to allocate memory to make the application crash. As an extra guarantee against your application's crashing, you might want to add some memory to this value. As indicated earlier in this chapter, 40 KB is a reasonable size for most applications.

```
CONST
    kMemCushion = 40 * 1024;           {size of memory cushion}
```

You should call the `IsMemoryAvailable` function before all nonessential memory requests, no matter how small. For example, suppose your application allocates a new, small relocatable block each time a user types a new line of text. That block might be small, but thousands of such blocks could take up a considerable amount of space. Therefore, you should check to see if there is sufficient memory available before allocating each one. (See Listing 5-6 on page 5-276 for an example of how to call `IsMemoryAvailable`.)

You should never, however, call the `IsMemoryAvailable` function before an essential memory request. When deciding how big to make the memory cushion for your application, you must make sure that essential requests can never deplete all of the cushion. Note that when you call the `IsMemoryAvailable` function for a nonessential

Introduction to Memory Management

request, essential requests might have already dipped into the memory cushion. In that case, `IsMemoryAvailable` returns `FALSE` no matter how small the nonessential request is.

Some actions should never be rejectable. For example, you should guarantee that there is always enough memory free to save open documents, and to perform typical maintenance tasks such as updating windows. Other user actions are likely to be always rejectable. For example, because you cannot allow the user to create an endless number of documents, you should make the New Document and Open Document menu commands rejectable.

Although the decisions of which actions to make rejectable are usually obvious, modal and modeless boxes present special problems. If you want to make such dialog boxes available at all costs, you must ensure that you allocate a large enough memory cushion to handle the maximum number of these dialog boxes that the user could open at once. If you consider a certain dialog box (for instance, a spelling checker) nonessential, you must be prepared to inform the user that there is not enough memory to open it if memory space become low.

Allocating Blocks of Memory

As you have seen, a key element of the memory-management scheme presented in this chapter is to disallow any nonessential memory allocation requests that would deplete the memory cushion. In practice, this means that, before calling `NewHandle`, `NewPtr`, or another function that allocates memory, you should check that the amount of space remaining after the allocation, if successful, exceeds the size of the memory cushion.

An easy way to do this is never to allocate memory for nonessential tasks by calling `NewHandle` or `NewPtr` directly. Instead call a function such as `NewHandleCushion`, defined in Listing 5-6, or `NewPtrCushion`, defined in Listing 5-7.

Listing 5-6 Allocating relocatable blocks

```
FUNCTION NewHandleCushion (logicalSize: Size): Handle;
BEGIN
    IF NOT IsMemoryAvailable(logicalSize) THEN
        NewHandleCushion := NIL
    ELSE
        BEGIN
            SetGrowZone(NIL);           {remove grow-zone function}
            NewHandleCushion := NewHandleClear(logicalSize);
            SetGrowZone(@MyGrowZone);  {install grow-zone function}
        END;
    END;
END;
```

The `NewHandleCushion` function first calls `IsMemoryAvailable` to determine whether allocating the requested number of bytes would deplete the memory cushion.

If so, `NewHandleCushion` returns `NIL` to indicate that the request has failed. Otherwise, if there is indeed sufficient space for the new block, `NewHandleCushion` calls `NewHandleClear` to allocate the relocatable block. Before calling `NewHandleClear`, however, `NewHandleCushion` disables the grow-zone function for the application heap. This prevents the grow-zone function from releasing any emergency memory reserve your application might be maintaining. See “Defining a Grow-Zone Function” on page 5-280 for details on grow-zone functions.

You can define a function `NewPtrCushion` to handle allocation of nonrelocatable blocks, as shown in Listing 5-7.

Listing 5-7 Allocating nonrelocatable blocks

```
FUNCTION NewPtrCushion (logicalSize: Size): Handle;
BEGIN
    IF NOT IsMemoryAvailable(logicalSize) THEN
        NewPtrCushion := NIL
    ELSE
        BEGIN
            SetGrowZone(NIL);           {remove grow-zone function}
            NewPtrCushion := NewPtrClear(logicalSize);
            SetGrowZone(@MyGrowZone);  {install grow-zone function}
        END;
    END;
END;
```

Note

The functions `NewHandleCushion` and `NewPtrCushion` allocate prezeroed blocks in your application heap. You can easily modify those functions if you do not want the blocks prezeroed. ♦

Listing 5-8 illustrates a typical way to call `NewPtrCushion`.

Listing 5-8 Allocating a dialog record

```
FUNCTION GetDialog (dialogID: Integer): DialogPtr;
VAR
    myPtr: Ptr;           {storage for the dialog record}
BEGIN
    myPtr := NewPtrCushion(SizeOf(DialogRecord));
    IF MemError = noErr THEN
        GetDialog := GetNewDialog(dialogID, myPtr, WindowPtr(-1))
    ELSE
        GetDialog := NIL;   {can't get memory}
    END;
END;
```

When you allocate memory directly, you can later release it by calling the `DisposeHandle` and `DisposePtr` procedures. When you allocate memory indirectly by calling a Toolbox routine, there is always a corresponding Toolbox routine to release that memory. For example, the `DisposeWindow` procedure releases memory allocated with the `NewWindow` function. Be sure to use these special Toolbox routines instead of the generic Memory Manager routines when applicable.

Maintaining a Memory Reserve

A simple way to help ensure that your application always has enough memory available for essential operations is to maintain an emergency memory reserve. This **memory reserve** is a block of memory that your application uses only for essential operations and only when all other heap space has been allocated. This section illustrates one way to implement a memory reserve in your application.

To create and maintain an emergency memory reserve, you follow three distinct steps:

- When your application starts up, you need to allocate a block of reserve memory. Because you allocate the block, it is no longer free in the heap and does not enter into the free-space determination done by `IsMemoryAvailable`.
- When your application needs to fulfill an essential memory request and there isn't enough space in your heap to satisfy the request, you can release the reserve. This effectively ensures that you always have the memory you request, at least for essential operations. You can use a grow-zone function to release the reserve when necessary; see “Defining a Grow-Zone Function” on page 5-280 for details.
- Each time through your main event loop, you should check whether the reserve has been released. If it has, you should attempt to recover the reserve. If you cannot recover the reserve, you should warn the user that memory is critically short.

To refer to the emergency reserve, you can declare a global variable of type `Handle`.

VAR

```
gEmergencyMemory: Handle; {handle to emergency memory reserve}
```

Listing 5-9 defines a function that you can call early in your application's execution (before entering your main event loop) to create an emergency memory reserve. This function also installs the application-defined grow-zone procedure. See “Defining a Grow-Zone Function” on page 5-280 for a description of the grow-zone function.

Listing 5-9 Creating an emergency memory reserve

```
PROCEDURE InitializeEmergencyMemory;
BEGIN
    gEmergencyMemory := NewHandle(kEmergencyMemorySize);
    SetGrowZone(@MyGrowZone);
END;
```

The `InitializeEmergencyMemory` procedure defined in Listing 5-9 simply allocates a relocatable block of a predefined size. That block is the emergency memory reserve. A reasonable size for the memory reserve is whatever size you use for the memory cushion. Once again, 40 KB is a good size for many applications.

```
CONST
    kEmergencyMemorySize = 40 * 1024; {size of memory reserve}
```

When using a memory reserve, you need to change the `IsMemoryAvailable` function defined earlier in Listing 5-5. You need to make sure, when determining whether a nonessential memory allocation request should be honored, that the memory reserve has not been released. To check that the memory reserve is intact, use the function `IsEmergencyMemory` defined in Listing 5-10.

Listing 5-10 Checking the emergency memory reserve

```
FUNCTION IsEmergencyMemory: Boolean;
BEGIN
    IsEmergencyMemory :=
        (gEmergencyMemory <> NIL) & (gEmergencyMemory^ <> NIL);
END;
```

Then, you can replace the function `IsMemoryAvailable` defined in Listing 5-5 (page 5-275) by the version defined in Listing 5-11.

Listing 5-11 Determining whether allocating memory would deplete the memory cushion

```
FUNCTION IsMemoryAvailable (memRequest: LongInt): Boolean;
VAR
    total: LongInt;    {total free memory if heap purged}
    contig: LongInt;   {largest contiguous block if heap purged}
BEGIN
    IF NOT IsEmergencyMemory THEN {is emergency memory available?}
        IsMemoryAvailable := FALSE
    ELSE
        BEGIN
            PurgeSpace(total, contig);
            IsMemoryAvailable := ((memRequest + kMemCushion) < contig);
        END;
    END;
```

As you can see, this is exactly like the earlier version except that it indicates that memory is not available if the memory reserve is not intact.

Introduction to Memory Management

Once you have allocated the memory reserve early in your application's execution, it should be released only to honor essential memory requests when there is no other space available in your heap. You can install a simple grow-zone function that takes care of releasing the reserve at the proper moment. Each time through your main event loop, you can check whether the reserve is still intact; to do this, add these lines of code to your main event loop, before you make your event call:

```
IF NOT IsEmergencyMemory THEN
    RecoverEmergencyMemory;
```

The `RecoverEmergencyMemory` function, defined in Listing 5-12, simply attempts to reallocate the memory reserve.

Listing 5-12 Reallocating the emergency memory reserve

```
PROCEDURE RecoverEmergencyMemory;
BEGIN
    ReallocateHandle(gEmergencyMemory, kEmergencyMemorySize);
END;
```

If you are unable to reallocate the memory reserve, you might want to notify the user that because memory is in short supply, steps should be taken to save any important data and to free some memory.

Defining a Grow-Zone Function

The Memory Manager calls your heap's grow-zone function only after other attempts to obtain enough memory to satisfy a memory allocation request have failed. A grow-zone function should be of the following form:

```
FUNCTION MyGrowZone (cbNeeded: Size): LongInt;
```

The Memory Manager passes to your function (in the `cbNeeded` parameter) the number of bytes it needs. Your function can do whatever it likes to free that much space in the heap. For example, your grow-zone function might dispose of certain blocks or make some unpurgeable blocks purgeable. Your function should return the number of bytes, if any, it managed to free.

When the function returns, the Memory Manager once again purges and compacts the heap and tries again to allocate the requested amount of memory. If there is still insufficient memory, the Memory Manager calls your grow-zone function again, but only if the function returned a nonzero value when last called. This mechanism allows your grow-zone function to release memory gradually; if the amount it releases is not enough, the Memory Manager calls it again and gives it the opportunity to take more drastic measures.

Typically a grow-zone function frees space by calling the `EmptyHandle` procedure, which purges a relocatable block from the heap and sets the block's master pointer to `NIL`. This is preferable to disposing of the space (by calling the `DisposeHandle` procedure), because you are likely to want to reallocate the block.

The Memory Manager might designate a particular relocatable block in the heap as **protected**; your grow-zone function should not move or purge that block. You can determine which block, if any, the Memory Manager has protected by calling the `GZSaveHnd` function in your grow-zone function.

Listing 5-13 defines a very basic grow-zone function. The `MyGrowZone` function attempts to create space in the application heap simply by releasing the block of emergency memory. First, however, it checks that (1) the emergency memory hasn't already been released and (2) the emergency memory is not a protected block of memory (as it would be, for example, during an attempt to reallocate the emergency memory block). If either of these conditions isn't true, then `MyGrowZone` returns 0 to indicate that no memory was released.

Listing 5-13 A grow-zone function that releases emergency storage

```
FUNCTION MyGrowZone (cbNeeded: Size): LongInt;
VAR
    theA5:    LongInt;                {value of A5 when function is called}
BEGIN
    theA5 := SetCurrentA5;            {remember current value of A5; install ours}
    IF (gEmergencyMemory^ <> NIL) & (gEmergencyMemory <> GZSaveHnd) THEN
        BEGIN
            EmptyHandle(gEmergencyMemory);
            MyGrowZone := kEmergencyMemorySize;
        END
    ELSE
        MyGrowZone := 0;              {no more memory to release}
        theA5 := SetA5(theA5);        {restore previous value of A5}
    END;
END;
```

The function `MyGrowZone` defined in Listing 5-13 saves the current value of the A5 register when it begins and then restores the previous value before it exits. This is necessary because your grow-zone function might be called at a time when the system is attempting to allocate memory and value in the A5 register is not correct. See the chapter “Memory Management Utilities” in this book for more information about saving and restoring the A5 register.

Note

You need to save and restore the A5 register only if your grow-zone function accesses your A5 world. (In Listing 5-13, the grow-zone function uses the global variable `gEmergencyMemory`.) ♦

Memory Management Reference

This section describes the routines used to illustrate the memory-management techniques presented earlier in this chapter. In particular, it describes the routines that allow you to manipulate blocks of memory in your application heap.

Note

For a complete description of all Memory Manager data types and routines, see the chapter “Memory Manager” in this book. ♦

Memory Management Routines

This section describes the routines you can use to set up your application’s heap, allocate and dispose of relocatable and nonrelocatable blocks, manipulate those blocks, assess the availability of memory in your application’s heap, free memory from the heap, and install a grow-zone function for your heap.

Note

The result codes listed for Memory Manager routines are usually not directly returned to your application. You need to call the `MemError` function (or, from assembly language, inspect the `MemErr` global variable) to get a routine’s result code. ♦

You cannot call most Memory Manager routines at interrupt time for several reasons. You cannot allocate memory at interrupt time because the Memory Manager might already be handling a memory-allocation request and the heap might be in an inconsistent state. More generally, you cannot call at interrupt time any Memory Manager routine that returns its result code via the `MemError` function, even if that routine doesn’t allocate or move memory. Resetting the `MemErr` global variable at interrupt time can lead to unexpected results if the interrupted code depends on the value of `MemErr`. Note that Memory Manager routines like `HLock` return their results via `MemError` and therefore should not be called in interrupt code.

Setting Up the Application Heap

The Operating System automatically initializes your application’s heap when your application is launched. To help prevent heap fragmentation, you should call the procedures in this section before you allocate any blocks of memory in your heap.

Use the `MaxApplZone` procedure to extend the application heap zone to the application heap limit so that the Memory Manager does not do so gradually as memory requests require. Use the `MoreMasters` procedure to preallocate enough blocks of master pointers so that the Memory Manager never needs to allocate new master pointer blocks for you.

MaxApplZone

To help ensure that you can use as much of the application heap zone as possible, call the `MaxApplZone` procedure. Call this once near the beginning of your program, after you have expanded your stack.

```
PROCEDURE MaxApplZone;
```

DESCRIPTION

The `MaxApplZone` procedure expands the application heap zone to the application heap limit. If you do not call `MaxApplZone`, the application heap zone grows as necessary to fulfill memory requests. The `MaxApplZone` procedure does not purge any blocks currently in the zone. If the zone already extends to the limit, `MaxApplZone` does nothing.

It is a good idea to call `MaxApplZone` once at the beginning of your program if you intend to maintain an effectively partitioned heap. If you do not call `MaxApplZone` and then call `MoveHHi` to move relocatable blocks to the top of the heap zone before locking them, the heap zone could later grow beyond these locked blocks to fulfill a memory request. If the Memory Manager were to allocate a nonrelocatable block in this new space, your heap would be fragmented.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `MaxApplZone` are

Registers on exit

D0 Result code

RESULT CODES

`noErr` 0 No error

MoreMasters

Call the `MoreMasters` procedure several times at the beginning of your program to prevent the Memory Manager from running out of master pointers in the middle of application execution. If it does run out, it allocates more, possibly causing heap fragmentation.

```
PROCEDURE MoreMasters;
```

DESCRIPTION

The `MoreMasters` procedure allocates another block of master pointers in the current heap zone. In the application heap, a block of master pointers consists of 64 master pointers, and in the system heap, a block consists of 32 master pointers. (These values, however, might change in future versions of system software.) When you initialize additional heap zones, you can specify the number of master pointers you want to have in a block of master pointers.

The Memory Manager automatically calls `MoreMasters` once for every new heap zone, including the application heap zone.

You should call `MoreMasters` at the beginning of your program enough times to ensure that the Memory Manager never needs to call it for you. For example, if your application never allocates more than 300 relocatable blocks in its heap zone, then five calls to the `MoreMasters` should be enough. It's better to call `MoreMasters` too many times than too few. For instance, if your application usually allocates about 100 relocatable blocks but might allocate 1000 in a particularly busy session, call `MoreMasters` enough times at the beginning of the program to accommodate times of greater memory use.

If you are forced to call `MoreMasters` so many times that it causes a significant slowdown, you could change the `moreMast` field of the zone header to the total number of master pointers you need and then call `MoreMasters` just once. Afterward, be sure to restore the `moreMast` field to its original value.

SPECIAL CONSIDERATIONS

Because `MoreMasters` allocates memory, you should not call it at interrupt time.

The calls to `MoreMasters` at the beginning of your application should be in the main code segment of your application or in a segment that the main segment never unloads.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `MoreMasters` are

Registers on exit

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory

GetApplLimit

Use the `GetApplLimit` function to get the application heap limit, beyond which the application heap cannot expand.

```
FUNCTION GetApplLimit: Ptr;
```

DESCRIPTION

The `GetApplLimit` function returns the current application heap limit. The Memory Manager expands the application heap only up to the byte preceding this limit.

Nothing prevents the stack from growing below the application limit. If the Operating System detects that the stack has crashed into the heap, it generates a system error. To avoid this, use `GetApplLimit` and the `SetApplLimit` procedure to set the application limit low enough so that a growing stack does not encounter the heap.

Note

The `GetApplLimit` function does not indicate the amount of memory available to your application. ♦

ASSEMBLY-LANGUAGE INFORMATION

The global variable `ApplLimit` contains the current application heap limit.

SetApplLimit

Use the `SetApplLimit` procedure to set the application heap limit, beyond which the application heap cannot expand.

```
PROCEDURE SetApplLimit (zoneLimit: Ptr);
```

zoneLimit A pointer to a byte in memory demarcating the upper boundary of the application heap zone. The zone can grow to include the byte preceding `zoneLimit` in memory, but no further.

DESCRIPTION

The `SetApplLimit` procedure sets the current application heap limit to `zoneLimit`. The Memory Manager then can expand the application heap only up to the byte

preceding the application limit. If the zone already extends beyond the specified limit, the Memory Manager does not cut it back but does prevent it from growing further.

Note

The `zoneLimit` parameter is not a byte count, but an absolute byte in memory. Thus, you should use the `SetApplLimit` procedure only with a value obtained from the Memory Manager functions `GetApplLimit` or `ApplicationZone`. ♦

You cannot change the limit of zones other than the application heap zone.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `SetApplLimit` are

Registers on entry

A0 Pointer to desired new zone limit

Registers on exit

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory

SEE ALSO

To use `SetApplLimit` to expand the default size of the stack, see the discussion in “Changing the Size of the Stack” on page 5-271.

Allocating and Releasing Relocatable Blocks of Memory

You can use the `NewHandle` function to allocate a relocatable block of memory. If you want to allocate new blocks of memory with their bits precleared to 0, you can use the `NewHandleClear` function.

▲ **WARNING**

You should not call any of these memory-allocation routines at interrupt time. ▲

You can use the `DisposeHandle` procedure to free relocatable blocks of memory you have allocated.

NewHandle

You can use the `NewHandle` function to allocate a relocatable memory block of a specified size.

```
FUNCTION NewHandle (logicalSize: Size): Handle;
```

`logicalSize`

The requested size (in bytes) of the relocatable block.

DESCRIPTION

The `NewHandle` function attempts to allocate a new relocatable block in the current heap zone with a logical size of `logicalSize` bytes and then return a handle to the block. The new block is unlocked and unpurgeable. If `NewHandle` cannot allocate a block of the requested size, it returns `NIL`.

▲ WARNING

Do not try to manufacture your own handles without this function by simply assigning the address of a variable of type `Ptr` to a variable of type `Handle`. The resulting “fake handle” would not reference a relocatable block and could cause a system crash. ▲

The `NewHandle` function pursues all available avenues to create a block of the requested size, including compacting the heap zone, increasing its size, and purging blocks from it. If all of these techniques fail and the heap zone has a grow-zone function installed, `NewHandle` calls the function. Then `NewHandle` tries again to free the necessary amount of memory, once more compacting and purging the heap zone if necessary. If memory still cannot be allocated, `NewHandle` calls the grow-zone function again, unless that function had returned 0, in which case `NewHandle` gives up and returns `NIL`.

SPECIAL CONSIDERATIONS

Because `NewHandle` allocates memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `NewHandle` are

Registers on entry

A0 Number of logical bytes requested

Registers on exit

A0 Address of the new block’s master pointer or
 `NIL`

D0 Result code

Introduction to Memory Management

If you want to clear the bytes of a block of memory to 0 when you allocate it with the `NewHandle` function, set bit 9 of the routine trap word. You can usually do this by supplying the word `CLEAR` as the second argument to the routine macro, as follows:

```
_NewHandle ,CLEAR
```

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory in heap zone

SEE ALSO

If you allocate a relocatable block that you plan to lock for long periods of time, you can prevent heap fragmentation by allocating the block as low as possible in the heap zone. To do this, see the description of the `ReserveMem` procedure on page 5-302.

If you plan to lock a relocatable block for short periods of time, you might want to move it to the top of the heap zone to prevent heap fragmentation. For more information, see the description of the `MoveHHI` procedure on page 5-303.

NewHandleClear

You can use the `NewHandleClear` function to allocate prezeroed memory in a relocatable block of a specified size.

```
FUNCTION NewHandleClear (logicalSize: Size): Handle;
```

`logicalSize`

The requested size (in bytes) of the relocatable block. The `NewHandleClear` function sets each of these bytes to 0.

DESCRIPTION

The `NewHandleClear` function works much as the `NewHandle` function does but sets all bytes in the new block to 0 instead of leaving the contents of the block undefined.

Currently, `NewHandleClear` clears the block one byte at a time. For a large block, it might be faster to clear the block manually a long word at a time.

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory in heap zone

DisposeHandle

When you are completely done with a relocatable block, call the `DisposeHandle` procedure to free it and its master pointer for other uses.

```
PROCEDURE DisposeHandle (h: Handle);
```

`h` A handle to a relocatable block.

DESCRIPTION

The `DisposeHandle` procedure releases the memory occupied by the relocatable block whose handle is `h`. It also frees the handle's master pointer for other uses.

▲ WARNING

After a call to `DisposeHandle`, all handles to the released block become invalid and should not be used again. Any subsequent calls to `DisposeHandle` using an invalid handle might damage the master pointer list. ▲

Do not use `DisposeHandle` to dispose of a handle obtained from the Resource Manager (for example, by a previous call to `GetResource`); use `ReleaseResource` instead. If, however, you have called `DetachResource` on a resource handle, you should dispose of the storage by calling `DisposeHandle`.

SPECIAL CONSIDERATIONS

Because `DisposeHandle` purges memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `DisposeHandle` are

Registers on entry

`A0` Handle to the relocatable block to be disposed of

Registers on exit

`D0` Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>memWZErr</code>	-111	Attempt to operate on a free block

Allocating and Releasing Nonrelocatable Blocks of Memory

You can use the `NewPtr` function to allocate a nonrelocatable block of memory. If you want to allocate new blocks of memory with their bits precleared to 0, you can use the `NewPtrClear` function.

▲ **WARNING**

You should not call any of these memory-allocation routines at interrupt time. ▲

You can use the `DisposePtr` procedure to free nonrelocatable blocks of memory you have allocated.

NewPtr

You can use the `NewPtr` function to allocate a nonrelocatable block of memory of a specified size.

```
FUNCTION NewPtr (logicalSize: Size): Ptr;
```

`logicalSize`

The requested size (in bytes) of the nonrelocatable block.

DESCRIPTION

The `NewPtr` function attempts to allocate, in the current heap zone, a nonrelocatable block with a logical size of `logicalSize` bytes and then return a pointer to the block. If the requested number of bytes cannot be allocated, `NewPtr` returns `NIL`.

The `NewPtr` function attempts to reserve space as low in the heap zone as possible for the new block. If it is able to reserve the requested amount of space, `NewPtr` allocates the nonrelocatable block in the gap `ReserveMem` creates. Otherwise, `NewPtr` returns `NIL` and generates a `memFullErr` error.

SPECIAL CONSIDERATIONS

Because `NewPtr` allocates memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `NewPtr` are

Registers on entry

A0 Number of logical bytes requested

Registers on exit

A0 Address of the new block or
 NIL

D0 Result code

If you want to clear the bytes of a block of memory to 0 when you allocate it with the `NewPtr` function, set bit 9 of the routine trap word. You can usually do this by supplying the word `CLEAR` as the second argument to the routine macro, as follows:

```
_NewPtr ,CLEAR
```

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory

NewPtrClear

You can use the `NewPtrClear` function to allocate prezeroed memory in a nonrelocatable block of a specified size.

```
FUNCTION NewPtrClear (logicalSize: Size): Ptr;
```

`logicalSize`

The requested size (in bytes) of the nonrelocatable block.

DESCRIPTION

The `NewPtrClear` function works much as the `NewPtr` function does, but sets all bytes in the new block to 0 instead of leaving the contents of the block undefined.

Currently, `NewPtrClear` clears the block one byte at a time. For a large block, it might be faster to clear the block manually a long word at a time.

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory

DisposePtr

When you are completely done with a nonrelocatable block, call the `DisposePtr` procedure to free it for other uses.

```
PROCEDURE DisposePtr (p: Ptr);
```

`p` A pointer to the nonrelocatable block you want to dispose of.

DESCRIPTION

The `DisposePtr` procedure releases the memory occupied by the nonrelocatable block specified by `p`.

▲ WARNING

After a call to `DisposePtr`, all pointers to the released block become invalid and should not be used again. Any subsequent use of a pointer to the released block might cause a system error. ▲

SPECIAL CONSIDERATIONS

Because `DisposePtr` purges memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `DisposePtr` are

Registers on entry

A0 Pointer to the nonrelocatable block to be disposed of

Registers on exit

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>memWZErr</code>	-111	Attempt to operate on a free block

Setting the Properties of Relocatable Blocks

A relocatable block can be either locked or unlocked and either purgeable or unpurgeable. In addition, it can have its resource bit either set or cleared. To determine the state of any of these properties, use the `HGetState` function. To change these

properties, use the `HLock`, `HUnlock`, `HPurge`, `HNoPurge`, `HSetRBit`, and `HClrRBit` procedures. To restore these properties, use the `HSetState` procedure.

▲ **WARNING**

Be sure to use these procedures to get and set the properties of relocatable blocks. In particular, do not rely on the structure of master pointers, because their structure in 24-bit mode is different from their structure in 32-bit mode. ▲

HGetState

You can use the `HGetState` function to get the current properties of a relocatable block (perhaps so that you can change and then later restore those properties).

```
FUNCTION HGetState (h: Handle): SignedByte;
```

`h` A handle to a relocatable block.

DESCRIPTION

The `HGetState` function returns a signed byte containing the flags of the master pointer for the given handle. You can save this byte, change the state of any of the flags, and then restore their original states by passing the byte to the `HSetState` procedure, described next.

You can use bit-manipulation functions on the returned signed byte to determine the value of a given attribute. Currently the following bits are used:

Bit	Meaning
0–4	Reserved
5	Set if relocatable block is a resource
6	Set if relocatable block is purgeable
7	Set if relocatable block is locked

If an error occurs during an attempt to get the state flags of the specified relocatable block, `HGetState` returns the low-order byte of the result code as its function result. For example, if the handle `h` points to a master pointer whose value is `NIL`, then the signed byte returned by `HGetState` will contain the value `-109`.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HGetState` are

Registers on entry

A0 Handle whose properties you want to get

Registers on exit

D0 Byte containing
 flags

RESULT CODES

<code>noErr</code>	0	No error
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memWZErr</code>	-111	Attempt to operate on a free block

HSetState

You can use the `HSetState` procedure to restore properties of a block after a call to `HGetState`.

```
PROCEDURE HSetState (h: Handle; flags: SignedByte);
```

`h` A handle to a relocatable block.

`flags` A signed byte specifying the properties to which you want to set the relocatable block.

DESCRIPTION

The `HSetState` procedure restores to the handle `h` the properties specified in the `flags` signed byte. See the description of the `HGetState` function for a list of the currently used bits in that byte. Because additional bits of the `flags` byte could become significant in future versions of system software, use `HSetState` only with a byte returned by `HGetState`. If you need to set two or three properties of a relocatable block at once, it is better to use the procedures that set individual properties than to manipulate the bits returned by `HGetState` and then call `HSetState`.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HSetState` are

Registers on entry

- A0 Handle whose properties you want to set
- D0 Byte containing flags indicating the handle's new properties

Registers on exit

- D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memWZErr</code>	-111	Attempt to operate on a free block

HLock

You can use the `HLock` procedure to lock a relocatable block so that it does not move in the heap. If you plan to dereference a handle and then allocate, move, or purge memory (or call a routine that does so), then you should lock the handle before using the dereferenced handle.

```
PROCEDURE HLock (h: Handle);
```

`h` A handle to a relocatable block.

DESCRIPTION

The `HLock` procedure locks the relocatable block to which `h` is a handle, preventing it from being moved within its heap zone. If the block is already locked, `HLock` does nothing.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HLock` are

Registers on entry

A0 Handle to lock

Registers on exit

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memWZErr</code>	-111	Attempt to operate on a free block

SEE ALSO

If you plan to lock a relocatable block for long periods of time, you can prevent fragmentation by ensuring that the block is as low as possible in the heap zone. To do this, see the description of the `ReserveMem` procedure on page 5-302.

If you plan to lock a relocatable block for short periods of time, you can prevent heap fragmentation by moving the block to the top of the heap zone before locking. For more information, see the description of the `MoveHHi` procedure on page 5-303.

HUnlock

You can use the `HUnlock` procedure to unlock a relocatable block so that it is free to move in its heap zone.

```
PROCEDURE HUnlock (h: Handle);
```

`h` A handle to a relocatable block.

DESCRIPTION

The `HUnlock` procedure unlocks the relocatable block to which `h` is a handle, allowing it to be moved within its heap zone. If the block is already unlocked, `HUnlock` does nothing.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HUnlock` are

Registers on entry

A0 Handle to unlock

Registers on exit

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memWZErr</code>	-111	Attempt to operate on a free block

HPurge

You can use the `HPurge` procedure to mark a relocatable block so that it can be purged if a memory request cannot be fulfilled after compaction.

```
PROCEDURE HPurge (h: Handle);
```

`h` A handle to a relocatable block.

DESCRIPTION

The `HPurge` procedure makes the relocatable block to which `h` is a handle purgeable. If the block is already purgeable, `HPurge` does nothing.

The Memory Manager might purge the block when it needs to purge the heap zone containing the block to satisfy a memory request. A direct call to the `PurgeMem` procedure or the `MaxMem` function would also purge blocks marked as purgeable.

Once you mark a relocatable block as purgeable, you should make sure that handles to the block are not empty before you access the block. If they are empty, you must reallocate space for the block and recopy the block's data from another source, such as a resource file, before using the information in the block.

If the block to which `h` is a handle is locked, `HPurge` does not unlock the block but does mark it as purgeable. If you later call `HUnlock` on `h`, the block is subject to purging.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HPurge` are

Registers on entry

A0 Handle to make purgeable

Registers on exit

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memWZErr</code>	-111	Attempt to operate on a free block

SEE ALSO

If the Memory Manager has purged a block, you can reallocate space for it by using the `ReallocateHandle` procedure, described on page 5-300.

You can immediately free the space taken by a handle without disposing of it by calling `EmptyHandle`. This procedure, described on page 5-299, does not require that the block be purgeable.

HNoPurge

You can use the `HNoPurge` procedure to mark a relocatable block so that it cannot be purged.

```
PROCEDURE HNoPurge (h: Handle);
```

`h` A handle to a relocatable block.

DESCRIPTION

The `HNoPurge` procedure makes the relocatable block to which `h` is a handle un purgeable. If the block is already un purgeable, `HNoPurge` does nothing.

The `HNoPurge` procedure does not reallocate memory for a handle if it has already been purged.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HNoPurge` are

Registers on entry

A0 Handle to make unpurgeable

Registers on exit

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memWZErr</code>	-111	Attempt to operate on a free block

SEE ALSO

If you want to reallocate memory for a relocatable block that has already been purged, you can use the `ReallocateHandle` procedure, described in the next section, “Managing Relocatable Blocks.”

Managing Relocatable Blocks

The Memory Manager provides routines that allow you to purge and later reallocate space for relocatable blocks and control where in their heap zone relocatable blocks are located.

To free the memory taken up by a relocatable block without releasing the master pointer to the block for other uses, use the `EmptyHandle` procedure. To reallocate space for a handle that you have emptied or the Memory Manager has purged, use the `ReallocateHandle` procedure.

To ensure that a relocatable block that you plan to lock for short or long periods of time does not cause heap fragmentation, use the `MoveHHI` and the `ReserveMem` procedures, respectively.

EmptyHandle

The `EmptyHandle` procedure allows you to free memory taken by a relocatable block without freeing the relocatable block’s master pointer for other uses.

```
PROCEDURE EmptyHandle (h: Handle);
```

`h` A handle to a relocatable block.

DESCRIPTION

The `EmptyHandle` procedure purges the relocatable block whose handle is `h` and sets the handle's master pointer to `NIL`. The block whose handle is `h` must be unlocked but need not be purgeable.

Note

If there are multiple handles to the relocatable block, then calling the `EmptyHandle` procedure empties them all, because all of the handles share a common master pointer. When you later use `ReallocateHandle` to reallocate space for the block, the master pointer is updated, and all of the handles reference the new block correctly. ♦

SPECIAL CONSIDERATIONS

Because `EmptyHandle` purges memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `EmptyHandle` are

Registers on entry

A0 Handle to relocatable block

Registers on exit

A0 Handle to relocatable block

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>memWZErr</code>	-111	Attempt to operate on a free block
<code>memPurErr</code>	-112	Attempt to purge a locked block

SEE ALSO

To free the memory taken up by a relocatable block and release the block's master pointer for other uses, use the `DisposeHandle` procedure, described on page 5-289.

ReallocateHandle

To recover space for a relocatable block that you have emptied or the Memory Manager has purged, use the `ReallocateHandle` procedure.

```
PROCEDURE ReallocateHandle (h: Handle; logicalSize: Size);
```

`h` A handle to a relocatable block.

`logicalSize`

The desired new logical size (in bytes) of the relocatable block.

DESCRIPTION

The `ReallocateHandle` procedure allocates a new relocatable block with a logical size of `logicalSize` bytes. It updates the handle `h` by setting its master pointer to point to the new block. The new block is unlocked and unpurgeable.

Usually you use `ReallocateHandle` to reallocate space for a block that you have emptied or the Memory Manager has purged. If the handle references an existing block, `ReallocateHandle` releases that block before creating a new one.

Note

To reallocate space for a resource that has been purged, you should call `LoadResource`, not `ReallocateHandle`. ♦

If many handles reference a single purged, relocatable block, you need to call `ReallocateHandle` on just one of them.

In case of an error, `ReallocateHandle` neither allocates a new block nor changes the master pointer to which handle `h` points.

SPECIAL CONSIDERATIONS

Because `ReallocateHandle` might purge and allocate memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `ReallocateHandle` are

Registers on entry

A0 Handle for new relocatable block

D0 Desired logical size, in bytes, of new block

Registers on exit

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>memROZErr</code>	-99	Heap zone is read-only
<code>memFullErr</code>	-108	Not enough memory
<code>memWZErr</code>	-111	Attempt to operate on a free block
<code>memPurErr</code>	-112	Attempt to purge a locked block

ReserveMem

Use the `ReserveMem` procedure when you allocate a relocatable block that you intend to lock for long periods of time. This helps prevent heap fragmentation because it reserves space for the block as close to the bottom of the heap as possible. Consistent use of `ReserveMem` for this purpose ensures that all locked, relocatable blocks and nonrelocatable blocks are together at the bottom of the heap zone and thus do not prevent unlocked relocatable blocks from moving about the zone.

```
PROCEDURE ReserveMem (cbNeeded: Size);
```

`cbNeeded` The number of bytes to reserve near the bottom of the heap.

DESCRIPTION

The `ReserveMem` procedure attempts to create free space for a block of `cbNeeded` contiguous logical bytes at the lowest possible position in the current heap zone. It pursues every available means of placing the block as close as possible to the bottom of the zone, including moving other relocatable blocks upward, expanding the zone (if possible), and purging blocks from it.

Because `ReserveMem` does not actually allocate the block, you must combine calls to `ReserveMem` with calls to the `NewHandle` function.

Do not use the `ReserveMem` procedure for a relocatable block you intend to lock for only a short period of time. If you do so and then allocate a nonrelocatable block above it, the relocatable block becomes trapped under the nonrelocatable block when you unlock that relocatable block.

Note

It isn't necessary to call `ReserveMem` to reserve space for a nonrelocatable block, because the `NewPtr` function calls it automatically. Also, you do not need to call `ReserveMem` to reserve memory before you load a locked resource into memory, because the Resource Manager calls `ReserveMem` automatically. ♦

SPECIAL CONSIDERATIONS

Because the `ReserveMem` procedure could move and purge memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `ReserveMem` are

Registers on entry

D0 Number of bytes to reserve

Registers on exit

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory

MoveHHi

If you plan to lock a relocatable block for a short period of time, use the `MoveHHi` procedure, which moves the block to the top of the heap and thus helps prevent heap fragmentation.

```
PROCEDURE MoveHHi (h: Handle);
```

`h` A handle to a relocatable block.

DESCRIPTION

The `MoveHHi` procedure attempts to move the relocatable block referenced by the handle `h` upward until it reaches a nonrelocatable block, a locked relocatable block, or the top of the heap.

▲ WARNING

If you call `MoveHHi` to move a handle to a resource that has its `resChanged` bit set, the Resource Manager updates the resource by using the `WriteResource` procedure to write the contents of the block to disk. If you want to avoid this behavior, call the Resource Manager procedure `SetResPurge(FALSE)` before you call `MoveHHi`, and then call `SetResPurge(TRUE)` to restore the default setting. ▲

By using the `MoveHHi` procedure on relocatable blocks you plan to allocate for short periods of time, you help prevent islands of immovable memory from accumulating in (and thus fragmenting) the heap.

Do not use the `MoveHHi` procedure to move blocks you plan to lock for long periods of time. The `MoveHHi` procedure moves such blocks to the top of the heap, perhaps preventing other blocks already at the top of the heap from moving down once they are unlocked. Instead, use the `ReserveMem` procedure before allocating such blocks, thus keeping them in the bottom partition of the heap, where they do not prevent relocatable blocks from moving.

If you frequently lock a block for short periods of time and find that calling `MoveHHi` each time slows down your application, you might consider leaving the block always locked and calling the `ReserveMem` procedure before allocating it.

Once you move a block to the top of the heap, be sure to lock it if you do not want the Memory Manager to move it back to the middle partition as soon as it can. (The `MoveHHi` procedure cannot move locked blocks; be sure to lock blocks after, not before, calling `MoveHHi`.)

Note

Using the `MoveHHi` procedure without taking other precautionary measures to prevent heap fragmentation is useless, because even one small nonrelocatable or locked relocatable block in the middle of the heap might prevent `MoveHHi` from moving blocks to the top of the heap. ♦

SPECIAL CONSIDERATIONS

Because the `MoveHHi` procedure moves memory, you should not call it at interrupt time.

Don't call `MoveHHi` on blocks in the system heap. Don't call `MoveHHi` from a desk accessory.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `MoveHHi` are

Registers on entry

A0 Handle to move

Registers on exit

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memLockedErr</code>	-117	Block is locked

HLockHi

You can use the `HLockHi` procedure to move a relocatable block to the top of the heap and lock it.

```
PROCEDURE HLockHi (h: Handle);
```

`h` A handle to a relocatable block.

DESCRIPTION

The `HLockHi` procedure attempts to move the relocatable block referenced by the handle `h` upward until it reaches a nonrelocatable block, a locked relocatable block, or the top of the heap. Then `HLockHi` locks the block.

The `HLockHi` procedure is simply a convenient replacement for the pair of procedures `MoveHHi` and `HLock`.

SPECIAL CONSIDERATIONS

Because the `HLockHi` procedure moves memory, you should not call it at interrupt time.

Don't call `HLockHi` on blocks in the system heap. Don't call `HLockHi` from a desk accessory.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HLockHi` are

Registers on entry

`A0` Handle to move and lock

Registers on exit

`D0` Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memWZErr</code>	-111	Attempt to operate on a free block
<code>memLockedErr</code>	-117	Block is locked

Manipulating Blocks of Memory

The Memory Manager provides a routine for copying blocks of memory referenced by pointers. To copy a block of memory to a nonrelocatable block, you can use the `BlockMove` procedure.

BlockMove

To copy a sequence of bytes from one location in memory to another, you can use the `BlockMove` procedure.

```
PROCEDURE BlockMove (sourcePtr, destPtr: Ptr; byteCount: Size);
```

`sourcePtr` The address of the first byte to copy.

`destPtr` The address of the first byte to copy to.

`byteCount` The number of bytes to copy. If the value of `byteCount` is 0, `BlockMove` does nothing.

DESCRIPTION

The `BlockMove` procedure moves a block of `byteCount` consecutive bytes from the address designated by `sourcePtr` to that designated by `destPtr`. It updates no pointers.

The `BlockMove` procedure works correctly even if the source and destination blocks overlap.

SPECIAL CONSIDERATIONS

You can safely call `BlockMove` at interrupt time. Even though it moves memory, `BlockMove` does not move relocatable blocks, but simply copies bytes.

The `BlockMove` procedure currently flushes the processor caches whenever the number of bytes to be moved is greater than 12. This behavior can adversely affect your application's performance. You might want to avoid calling `BlockMove` to move small amounts of data in memory if there is no possibility of moving stale data or instructions. For more information about stale data and instructions, see the discussion of the processor caches in the chapter "Memory Management Utilities" in this book.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `BlockMove` are

Registers on entry

A0 Pointer to source

A1 Pointer to destination

D0 Number of bytes to copy

Registers on exit

D0 Result code

RESULT CODE

noErr 0 No error

Assessing Memory Conditions

The Memory Manager provides routines to test how much memory is available. To determine the total amount of free space in the current heap zone or the size of the maximum block that could be obtained after a purge of the heap, call the `PurgeSpace` function.

To find out whether a Memory Manager operation finished successfully, use the `MemError` function.

PurgeSpace

Use the `PurgeSpace` procedure to determine the total amount of free memory and the size of the largest allocatable block after a purge of the heap.

```
PROCEDURE PurgeSpace (VAR total: LongInt; VAR contig: LongInt);
```

<code>total</code>	On exit, the total amount of free memory in the current heap zone if it were purged.
<code>contig</code>	On exit, the size of the largest contiguous block of free memory in the current heap zone if it were purged.

DESCRIPTION

The `PurgeSpace` procedure returns, in the `total` parameter, the total amount of space (in bytes) that could be obtained after a general purge of the current heap zone; this amount includes space that is already free. In the `contig` parameter, `PurgeSpace` returns the size of the largest allocatable block in the current heap zone that could be obtained after a purge of the zone.

The `PurgeSpace` procedure does not actually purge the current heap zone.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `PurgeSpace` are

Registers on exit

A0	Maximum number of contiguous bytes after purge
D0	Total free memory after purge

RESULT CODES

noErr 0 No error

MemError

To find out whether your application's last direct call to a Memory Manager routine executed successfully, use the `MemError` function.

```
FUNCTION MemError: OSErr;
```

DESCRIPTION

The `MemError` function returns the result code produced by the last Memory Manager routine your application called directly.

This function is useful during application debugging. You might also use the function as one part of a memory-management scheme to identify instances in which the Memory Manager rejects overly large memory requests by returning the error code `memFullErr`.

▲ WARNING

Do not rely on the `MemError` function as the only component of a memory-management scheme. For example, suppose you call `NewHandle` or `NewPtr` and receive the result code `noErr`, indicating that the Memory Manager was able to allocate sufficient memory. In this case, you have no guarantee that the allocation did not deplete your application's memory reserves to levels so low that simple operations might cause your application to crash. Instead of relying on `MemError`, check before making a memory request that there is enough memory both to fulfill the request and to support essential operations. ▲

ASSEMBLY-LANGUAGE INFORMATION

Because most Memory Manager routines return a result code in register D0, you do not ordinarily need to call the `MemError` function if you program in assembly language. See the description of an individual routine to find out whether it returns a result code in register D0. If not, you can examine the global variable `MemErr`. When `MemError` returns, register D0 contains the result code.

Registers on exit

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Error in parameter list
<code>memROZErr</code>	-99	Operation on a read-only zone
<code>memFullErr</code>	-108	Not enough memory
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memWZErr</code>	-111	Attempt to operate on a free block
<code>memPurErr</code>	-112	Attempt to purge a locked block
<code>memBCErr</code>	-115	Block check failed
<code>memLockedErr</code>	-117	Block is locked

Grow-Zone Operations

You can implement a grow-zone function that the Memory Manager calls when it cannot fulfill a memory request. You should use the grow-zone function only as a last resort to free memory when all else fails.

The `SetGrowZone` procedure specifies which function the Memory Manager should use for the current zone. The grow-zone function should call the `GZSaveHnd` function to receive a handle to a relocatable block that the grow-zone function must not move or purge.

SetGrowZone

To specify a grow-zone function for the current heap zone, pass a pointer to that function to the `SetGrowZone` procedure. Ordinarily, you call this procedure early in the execution of your application.

If you initialize your own heap zones besides the application and system zones, you can alternatively specify a grow-zone function as a parameter to the `InitZone` procedure.

```
PROCEDURE SetGrowZone (growZone: ProcPtr);
```

`growZone` A pointer to the grow-zone function.

DESCRIPTION

The `SetGrowZone` procedure sets the current heap zone's grow-zone function as designated by the `growZone` parameter. A `NIL` parameter value removes any grow-zone function the zone might previously have had.

The Memory Manager calls the grow-zone function only after exhausting all other avenues of satisfying a memory request, including compacting the zone, increasing its size (if it is the original application zone and is not yet at its maximum size), and purging blocks from it.

See "Grow-Zone Functions" on page 5-312 for a complete description of a grow-zone function.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `SetGrowZone` are

Registers on entry

A0 Pointer to new grow-zone function

Registers on exit

D0 Result code

RESULT CODES

noErr 0 No error

SEE ALSO

See “Defining a Grow-Zone Function” on page 5-280 for a description of a grow-zone function.

GZSaveHnd

Your grow-zone function must call the `GZSaveHnd` function to obtain a handle to a protected relocatable block that the grow-zone function must not move, purge, or delete.

```
FUNCTION GZSaveHnd: Handle;
```

DESCRIPTION

The `GZSaveHnd` function returns a handle to a relocatable block that the grow-zone function must not move, purge, or delete. It returns `NIL` if there is no such block. The returned handle is a handle to the block of memory being manipulated by the Memory Manager at the time that the grow-zone function is called.

ASSEMBLY-LANGUAGE INFORMATION

You can find the same handle in the global variable `GZRootHnd`.

Setting and Restoring the A5 Register

Any code that runs asynchronously or as a callback routine and that accesses the calling application’s A5 world must ensure that the A5 register correctly points to the boundary between the application parameters and the application global variables. To accomplish this, you can call the `SetCurrentA5` function at the beginning of any asynchronous or callback code that isn’t executed at interrupt time. If the code is executed at interrupt time, you must use the `SetA5` function to set the value of the A5 register. (You determine this value at noninterrupt time by calling `SetCurrentA5`.) Then you must restore the A5 register to its previous value before the interrupt code returns.

SetCurrentA5

You can use the `SetCurrentA5` function to get the current value of the system global variable `CurrentA5`.

```
FUNCTION SetCurrentA5: LongInt;
```

DESCRIPTION

The `SetCurrentA5` function does two things: First, it gets the current value in the A5 register and returns it to your application. Second, `SetCurrentA5` sets register A5 to the value of the low-memory global variable `CurrentA5`. This variable points to the boundary between the parameters and global variables of the current application.

SPECIAL CONSIDERATIONS

You cannot reliably call `SetCurrentA5` in code that is executed at interrupt time unless you first guarantee that your application is the current process (for example, by calling the Process Manager function `GetCurrentProcess`). In general, you should call `SetCurrentA5` at noninterrupt time and then pass the returned value to the interrupt code.

ASSEMBLY-LANGUAGE INFORMATION

You can access the value of the current application's A5 register with the low-memory global variable `CurrentA5`.

SetA5

In interrupt code that accesses application global variables, use the `SetA5` function first to restore a value previously saved using `SetCurrentA5`, and then, at the end of the code, to restore the A5 register to the value it had before the first call to `SetA5`.

```
FUNCTION SetA5 (newA5: LongInt): LongInt;
```

`newA5` The value to which the A5 register is to be changed.

DESCRIPTION

The `SetA5` function performs two tasks: it returns the address in the A5 register when the function is called, and it sets the A5 register to the address specified in `newA5`.

Application-Defined Routines

The techniques illustrated in this chapter use only one application-defined routine, a grow-zone function.

Grow-Zone Functions

The Memory Manager calls your application's grow-zone function whenever it cannot find enough contiguous memory to satisfy a memory allocation request and has exhausted other means of obtaining the space.

MyGrowZone

A grow-zone function should have the following form:

```
FUNCTION MyGrowZone (cbNeeded: Size): LongInt;
```

cbNeeded The physical size, in bytes, of the needed block, including the block header. The grow-zone function should attempt to create a free block of at least this size.

DESCRIPTION

Whenever the Memory Manager has exhausted all available means of creating space within your application heap—including purging, compacting, and (if possible) expanding the heap—it calls your application-defined grow-zone function. The grow-zone function can do whatever is necessary to create free space in the heap. Typically, a grow-zone function marks some unneeded blocks as purgeable or releases an emergency memory reserve maintained by your application.

The grow-zone function should return a nonzero value equal to the number of bytes of memory it has freed, or zero if it is unable to free any. When the function returns a nonzero value, the Memory Manager once again purges and compacts the heap zone and tries to reallocate memory. If there is still insufficient memory, the Memory Manager calls the grow-zone function again (but only if the function returned a nonzero value the previous time it was called). This mechanism allows your grow-zone function to release just a little bit of memory at a time. If the amount it releases at any time is not enough, the Memory Manager calls it again and gives it the opportunity to take more drastic measures.

The Memory Manager might designate a particular relocatable block in the heap as protected; your grow-zone function should not move or purge that block. You can determine which block, if any, the Memory Manager has protected by calling the `GZSaveHnd` function in your grow-zone function.

Remember that a grow-zone function is called while the Memory Manager is attempting to allocate memory. As a result, your grow-zone function should not allocate memory itself or perform any other actions that might indirectly cause memory to be allocated (such as calling routines in unloaded code segments or displaying dialog boxes).

You install a grow-zone function by passing its address to the `InitZone` procedure when you create a new heap zone or by calling the `SetGrowZone` procedure at any other time.

SPECIAL CONSIDERATIONS

Your grow-zone function might be called at a time when the system is attempting to allocate memory and the value in the A5 register is not correct. If your function accesses your application's A5 world or makes any trap calls, you need to set up and later restore the A5 register by calling `SetCurrentA5` and `SetA5`.

Because of the optimizations performed by some compilers, the actual work of the grow-zone function and the setting and restoring of the A5 register might have to be placed in separate procedures.

SEE ALSO

See "Defining a Grow-Zone Function" on page 5-280 for a definition of a sample grow-zone function.

Result Codes

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Error in parameter list
<code>memROZErr</code>	-99	Heap zone is read-only
<code>memFullErr</code>	-108	Not enough memory
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memWZErr</code>	-111	Attempt to operate on a free block
<code>memPurErr</code>	-112	Attempt to purge a locked block
<code>memBCErr</code>	-115	Block check failed
<code>memLockedErr</code>	-117	Block is locked

Introduction to Memory Management

Memory Manager

This chapter describes how your application can use the Memory Manager to manage memory both in its own partition and outside its partition. Ordinarily, you allocate memory in your application heap only. You might, however, occasionally need to access memory outside of your application partition, or you might want to create additional heap zones within your application partition.

You need to read this chapter if you want to use Memory Manager routines other than those described in the chapter “Introduction to Memory Management” in this book. That chapter shows how to use the Memory Manager and other system software components to perform the most common memory-manipulation operations while avoiding heap fragmentation and low memory situations. This chapter addresses a number of other important memory-related issues.

This chapter begins with a description of areas of memory that are outside your application’s partition and their typical uses. Then it describes how you can

- allocate temporary memory
- allocate memory in and install code into the system heap
- read and change the values of system global variables
- allocate high memory during the startup process
- create additional heap zones within your application’s partition
- install a purge-warning procedure for a heap zone

This chapter also addresses some advanced topics that are generally of use only to developers of very specialized applications or memory utilities. These advanced topics include

- how the Memory Manager organizes heap zones
- how the Memory Manager organizes memory blocks

To use this chapter, you should be familiar with ordinary use of the Memory Manager and other system software components that allow you to manage memory, as described in the chapter “Introduction to Memory Management” earlier in this book.

The “Memory Manager Reference” and “Result Codes” sections in this chapter provide a complete reference and summary of the constants, data types, and routines provided by the Memory Manager.

About the Memory Manager

The Memory Manager is the part of the Macintosh Operating System that controls the dynamic **allocation** of memory space. Ordinarily, you need to access information only within your own application’s heap, stack, and A5 world. Occasionally, however, you might need to use the Memory Manager to allocate temporary memory outside of your application’s partition or to initialize new heap zones within your application partition. You might also need to read a system global variable to obtain information about the environment in which your application is executing.

The Memory Manager provides a large number of routines that you can use to perform various operations on blocks within your application partition. You can use the Memory Manager to

- set up your application partition
- allocate and release both relocatable and nonrelocatable blocks in your application heap
- copy data from nonrelocatable blocks to relocatable blocks, and vice versa
- determine how much space is free in your heap
- determine the location of the top of your stack
- determine the size of a memory block and, if necessary, change that size
- change the properties of relocatable blocks
- install or remove a grow-zone function for your heap
- obtain the result code of the most recent Memory Manager routine executed

The Memory Manager also provides routines that you can use to access areas of memory outside your application partition. You can use the Memory Manager to

- allocate memory outside your partition that is currently unused by any open application or by the Operating System
- allocate memory in the system heap

This section describes the areas of memory that lie outside your application partition. It also describes multiple heap zones.

Temporary Memory

In the Macintosh multitasking environment, your application is limited to a particular memory partition (whose size is determined by information in the ‘SIZE’ resource of your application). The size of your application’s partition places certain limits on the size

of your application heap and hence on the sizes of the buffers and other data structures that your application can use.

If for some reason you need more memory than is currently available in your application heap, you can ask the Operating System to let you use any available memory that is not yet allocated to any other application. This memory, called **temporary memory**, is allocated from the available unused RAM; in general, that memory is not contiguous with the memory in your application's zone

Your application should use temporary memory only for occasional short-term purposes that could be accomplished in less space, though perhaps less efficiently. For example, if you want to copy a large file, you might try to allocate a fairly large buffer of temporary memory. If you receive the temporary memory, you can use the large buffer to copy data from the source file into the destination file. If, however, the request for temporary memory fails, you can instead use a smaller buffer within your application heap. Although the use of a smaller buffer might prolong the copy operation, the file is nonetheless copied.

One good reason for using temporary memory only occasionally is that you cannot assume that you will always receive the temporary memory you request. For example, if two or more applications use all available memory outside the system partition, then a request by any of them for some temporary memory would fail.

Another strategy for using temporary memory is to use it, when possible, for all nonessential memory requests. For example, you could allocate window records and any associated window data using temporary memory. This scheme allows you to keep your application partition relatively small (because you don't need space for nonessential tasks) but assumes that users will not fill up the available memory with other applications.

Multiple Heap Zones

A **heap zone** is a heap (that is, an area in which you can dynamically allocate and release memory on demand) together with a zone header and a zone trailer. The **zone header** is an area of memory that contains essential information about the heap, such as the number of free bytes in the heap and the addresses of the heap's grow-zone function and purge-warning procedure. The **zone trailer** is just a minimum-sized block placed as a marker at the end of the heap zone. (See "Heap Zones" on page 6-332 for a complete description of zone headers and trailers.)

When your application is executing, there exist at least two heap zones: your application's heap zone (created when your application was launched) and the system heap zone (created when the system was started up). The **system heap zone** is the heap zone that contains the system heap. Your **application heap zone** (also known as the **original application heap zone**) is the heap zone initially provided by the Memory Manager for use by your application and any system software routines your application calls.

Ordinarily, you allocate and release blocks of memory in the **current heap zone**, which by default is your application heap zone. Unless you change the current heap zone (for

Memory Manager

example, by calling the `InitZone` or `SetZone` procedures), you do not need to worry about which is the current zone; all blocks that you access are taken from the current heap zone, that is, your application heap zone.

Occasionally, however, you might need to allocate memory in the system heap zone. System software uses the system heap to store information it needs. Although, in general, you should not allocate memory in the system heap, there are several valid reasons for doing so. First, if you are implementing a system extension, the extension can use the system heap to store information. Second, if you want the Time Manager or Vertical Retrace Manager to execute some interrupt code when your application is not the current application, you might in certain cases need to store the task record and the task code in the system heap. Third, if you write interrupt code that itself uses heap memory, you should either place that memory in the system heap or hold it in real RAM to prevent page faults at interrupt time, as discussed in the chapter “Virtual Memory Manager” in this book.

You can create additional heap zones for your application’s own use by calling the `InitZone` procedure. If you do maintain more than one heap zone, you can find out which heap zone is the current one at any time by calling the `GetZone` function, and you can switch zones by calling the `SetZone` procedure. Almost all Memory Manager operations implicitly apply to the current heap zone. To refer to the system heap zone or to the (original) application heap zone, you can call the functions `SystemZone` or `ApplicationZone`. To find out which zone a particular block resides in, you can call the `HandleZone` function (if the block is relocatable) or the `PtrZone` function (if it’s nonrelocatable).

▲ **WARNING**

Be sure, when calling routines that access blocks, that the zone in which the block is located is the current zone. If, for example, you attempt to release an empty resource in the system zone when the current zone is not the system zone, the Operating System might incorrectly update the list of free master pointers in your partition. ▲

Once you have created a heap zone, it remains fixed in size and location. For this reason, it usually makes more sense to use the undivided application heap zone for all of your memory-allocation needs. You might, however, choose to initialize an additional heap zone in circumstances like these:

- If you are implementing a software development environment and want to launch applications within the development environment’s partition, you can initialize a heap zone for the launched application to use as its heap zone.
- If you want to avoid heap fragmentation but cannot prevent allocation of small nonrelocatable blocks in the middle of your program’s execution, you could, soon after your application starts up, allocate a small heap zone to hold the nonrelocatable blocks you allocate during execution.
- If you need to resize a particular handle quite often, you can minimize the resizing time by creating a heap zone whose size is set to the maximum size the handle will ever be assigned. Because there is only one relocatable block in the new heap zone, the

resizing is likely to happen more quickly than if that block were in the original heap zone (where other relocatable blocks in the zone might need to be moved).

Before deciding to create additional heap zones, however, make sure that you really need to. Maintaining multiple heap zones requires a considerable amount of extra work. You must always make sure to allocate or release memory in the correct zone, and you must monitor memory conditions in each zone so that your application doesn't run out of memory.

The System Global Variables

Just as the Toolbox stores information about your drawing environment in a set of QuickDraw global variables within your application partition, the Operating System and Toolbox store information about the entire multiple-application environment in a set of **system global variables**, also called low-memory global variables. The system global variables are stored in the lowest part of the physical RAM, in the system partition.

Most system global variables are intended for use by system software only, and you should never need to read or write them directly. Current versions of system software contain functions that return values equivalent to most of the important system global variables. Use those routines whenever they are available. However, you might occasionally need to access the value of a system global variable to maintain compatibility with previous versions of system software, or you might need to access a system global variable whose value no equivalent function returns.

The MPW interface file `SysEqu.p` defines the memory locations at which system global variables are stored in the latest version of system software. For example, `SysEqu.p` contains lines like these:

```
CONST
    RndSeed      = $156;  {random number seed (long)}
    Ticks        = $16A;  {ticks since last boot (unsigned long)}
    DeskHook     = $A6C;  {hook for painting desktop (pointer)}
    MBarHeight   = $BAA;  {height of menu bar (integer)}
```

You can use these memory locations to examine the value of one of these variables. See “Reading and Writing System Global Variables” on page 6-320 for instructions on reading and writing the values of system global variables from a high-level language.

You should avoid relying on the value of a system global variable whenever possible. The meanings of many global variables have changed in the past and will change again in the future. Using the system global variables documented in *Inside Macintosh* is fairly safe, but you risk incompatibility with future versions of system software if you attempt to access global variables defined in the interface files but not explicitly documented.

Even when *Inside Macintosh* does document a particular system global variable, you should use any available routines to access that variable's value instead of examining it directly. For example, you should use the `TickCount` function to find the number of ticks since startup instead of examining the `Ticks` global variable directly.

IMPORTANT

You should read or write the value of a system global variable only when that variable is documented in *Inside Macintosh* and when there is no alternate method of reading or writing the information you need. ▲

Using the Memory Manager

This section discusses the techniques you can use both to deal with memory outside of your application's partition and to manipulate your own application's partition.

You can use the techniques in this section to

- read and write the values of system global variables when there is no Toolbox routine that would accomplish the work for you
- check for the availability of temporary memory and use it to speed operations that depend on memory buffers
- allocate memory in the system heap
- install code into the system heap
- allocate memory at the high end of the available RAM from within a system extension during the startup process
- initialize new heap zones within your application heap zone, on your application's stack, or in the application global variables area
- install a purge-warning procedure for your application heap zone

Reading and Writing System Global Variables

In general, you should avoid relying on the values of system global variables whenever possible. However, you might occasionally need to access the value of one of these variables. Because the actual values associated with global variables in MPW's `SysEqu.p` interface file are memory locations, you can access the value of a low-memory variable simply by dereferencing a memory location.

Many system global variables are process-independent, but some are process-specific. The Operating System swaps the values of the process-specific variables as it switches processes. If you write interrupt code that reads low memory, that code could execute at a time when another process's system global variables are installed. Therefore, before reading low memory from interrupt code, you should call the Process Manager to ensure that your process is the current process. If it is not, you should not rely on the value of system global variables that could conceivably be process-specific.

Note

No available documentation distinguishes process-specific from process-independent system global variables. ♦

The routine defined in Listing 6-1 illustrates how you can read a system global variable, in this case the system global variable `BufPtr`, which gives the address of the highest byte of allocatable memory.

Listing 6-1 Reading the value of a system global variable

```
FUNCTION FindHighestByte: LongInt;
TYPE
    LongPtr = ^LongInt;
BEGIN
    FindHighestByte := LongPtr(BufPtr)^;
END;
```

In Pascal, the main technique for reading system global variables is to define a new data type that points to the variable type you want to read. In this example, the address is stored as a long integer. Thus, the memory location `BufPtr` is really a pointer to a long integer. Because of Pascal's strict typing rules, you must cast the low-memory address into a pointer to a long integer. Then, you can dereference the pointer and return the long integer itself as the function result.

You can use a similar technique to change the value of a system global variable. For example, suppose you are writing an extension that displays a window at startup time. To maintain compatibility with pre-Macintosh II systems, you need to clear the system global variable named `DeskHook`. This global variable holds a `ProcPtr` that references a procedure called by system software to paint the desktop. If the value of the pointer is `NIL`, the system software uses the standard desktop pattern. If you do not set `DeskHook` to `NIL`, the system software might attempt to use whatever random data it contains to call an updating procedure when you move or close your window. The procedure defined in Listing 6-2 sets `DeskHook` to `NIL`.

Listing 6-2 Changing the value of a system global variable

```
PROCEDURE ClearDeskHook;
TYPE
    ProcPtrPtr = ^ProcPtr;           {pointer to ProcPtr}
VAR
    deskHookProc: ProcPtrPtr;
BEGIN
    deskHookProc := ProcPtrPtr(DeskHook); {initialize variable}
    deskHookProc^ := NIL;               {clear DeskHook proc}
END;
```

You can use a similar technique to change the value of any other documented system global variable.

Extending an Application's Memory

Rather than using your application's 'SIZE' resource to specify a preferred partition size that is large enough to contain the largest possible application heap, you should specify a smaller but adequate partition size. When you need more memory for temporary use, you can use a set of Memory Manager routines for the allocation of temporary memory.

By using the routines for allocating temporary memory, your application can request some additional memory for occasional short-term needs. For example, the Finder uses these temporary-memory routines to secure buffer space for use during file copy operations. Any available memory (that is, memory currently unallocated to any application's partition) is dedicated to this purpose. The Finder releases this memory as soon as the copy is completed, thus making the memory available to other applications or to the Operating System for launching new applications.

Because the requested amount of memory might not be available, you cannot be sure that every request for temporary memory will be honored. Thus, you should make sure that your application will work even if your request for temporary memory is denied. For example, if the Finder cannot allocate a large temporary copy buffer, it uses a reserved small copy buffer from within its own heap zone, prolonging the copying but performing it nonetheless.

Temporary memory is taken from RAM that is reserved for (but not yet used by) other applications. Thus, if you use too much temporary memory or hold temporary memory for long periods of time, you might prevent the user from being able to launch other applications. In certain circumstances, however, you can hold temporary memory indefinitely. For example, if the temporary memory is used for open files and the user can free that memory simply by closing those files, it is safe to hold onto that memory as long as necessary.

Temporary memory is tracked (or monitored) for each application, and so you must use it only for code that is running on an application's behalf. Moreover, the Operating System frees all temporary memory allocated to an application when the application quits or crashes. As a result, you should not use temporary memory for VBL tasks, Time Manager tasks, or other procedures that should continue to be executed after your application quits. Similarly, it is wise not to use temporary memory for an interprocess buffer (that is, a buffer whose address is passed to another application in a high-level event) because the originating application could crash, quit, or be terminated, thereby causing the temporary memory to be released before (or even while) the receiving application uses that memory.

Although you can usually perform ordinary Memory Manager operations on temporary memory, there are two restrictions. First, you must never lock temporary memory across calls to `GetNextEvent` or `WaitNextEvent`. Second, although you can determine the zone from which temporary memory is generated (using the `HandleZone` function), you should not use this information to make new blocks or perform heap operations on your own.

Allocating Temporary Memory

You can request a block of memory for temporary use by calling the Memory Manager's `TempNewHandle` function. This function attempts to allocate a new relocatable block of the specified size for temporary use. For example, to request a block that is one-quarter megabyte in size, you might issue this command:

```
myHandle := TempNewHandle($40000, myErr); {request temp memory}
```

If the routine succeeds, it returns a handle to the block of memory. The block of memory returned by a successful call to `TempNewHandle` is initially unlocked. If an error occurs and `TempNewHandle` fails, it returns a `NIL` handle. You should always check for `NIL` handles before using any temporary memory. If you detect a `NIL` handle, the second parameter (in this example, `myErr`) contains the result code from the function.

Instead of asking for a specific amount of memory and then checking the returned handle to find out whether it was allocated, you might prefer to determine beforehand how much temporary memory is available. There are two functions that return information on the amount of free memory available for temporary allocation. The first is the `TempFreeMem` function, which you can use as follows:

```
memFree := TempFreeMem;      {find amount of free temporary memory}
```

The result is a long integer containing the amount, in bytes, of free memory available for temporary allocation. It usually isn't possible to allocate a block of this size because of fragmentation. Consequently, you'll probably want to use the second function, `TempMaxMem`, to determine the size of the largest contiguous block of space available. To allocate that block, you can write

```
mySize := TempMaxMem(grow);
myHandle := TempNewHandle(mySize, myErr);
```

The `TempMaxMem` function returns the size, in bytes, of the largest contiguous free block available for temporary allocation. (The `TempMaxMem` function is analogous to the `MaxMem` function.) The `grow` parameter is a variable parameter of type `Size`; after the function returns, it always contains 0, because the temporary memory does not come from the application's heap. Even when you use `TempMaxMem` to determine the size of the available memory, you should check that the handle returned by `TempNewHandle` is not `NIL`.

Determining the Features of Temporary Memory

Only computers running system software version 7.0 and later can use temporary memory as described in this chapter. For this reason, you should always check that the routines are available and that they have the features you require before calling them.

Memory Manager

Note

The temporary-memory routines are available in some earlier system software versions when MultiFinder is running. However, the handles to blocks of temporary memory are neither tracked nor real. ♦

The `Gestalt` function includes a selector to determine whether the temporary-memory routines are present in the operating environment and, if they are, whether the temporary-memory handles are tracked and whether they are real. If temporary-memory handles are not tracked, you must release temporary memory before your next call to `GetNextEvent` or `WaitNextEvent`. If temporary-memory handles are not real, then you cannot use normal Memory Manager routines such as `HLock` to manipulate them.

To determine whether the temporary-memory routines are implemented, you can check the value returned by the `TempMemCallsAvailable` function, defined in Listing 6-3.

Listing 6-3 Determining whether temporary-memory routines are available

```
FUNCTION TempMemCallsAvailable: Boolean;
VAR
    myErr:   OSErr;           {Gestalt result code}
    myRsp:   LongInt;         {response returned by Gestalt}
BEGIN
    TempMemCallsAvailable := FALSE;
    myErr := Gestalt(gestaltOSAttr, myRsp);
    IF myErr <> noErr THEN
        DoError(myErr)        {Gestalt failed}
    ELSE
        {check bit for temp mem support}
        TempMemCallsAvailable :=
            BAND(myRsp, gestaltTempMemSupport) <> 0;
END;
```

You can use similar code to determine whether temporary-memory handles are real and whether the temporary memory is tracked.

Using the System Heap

The system heap is used to store most of the information needed by the Operating System and other system software components. As a result, it is ideal for storing information needed by a system extension (which by definition extends the capabilities of system software). You might also need to use the system heap to store a task record and the code for an interrupt task that should continue to be executed when your application is not the current application.

Allocating blocks in the system heap is straightforward. Most ordinary Memory Manager routines have counterparts that allocate memory in the system heap zone instead of the current heap zone. For example, the counterpart of the `NewPtr` function is the

Memory Manager

`NewPtrSys` function. The following line of code allocates a new nonrelocatable block of memory in the system heap to store a Time Manager task record:

```
myTaskPtr := QElemPtr(NewPtrSys(SizeOf(TMTask)));
```

Alternatively, you can change the current zone and use ordinary Memory Manager operations, as follows:

```
SetZone(SystemZone);
myTaskPtr := QElemPtr(NewPtr(SizeOf(TMTask)));
...
SetZone(ApplicationZone);
```

You might also need to store the interrupt code itself in the system heap. For example, when an application that installed a vertical retrace task with the `VInstall` function is in the background, the Vertical Retrace Manager executes the task only if the `vblAddr` field of the task record points to a routine in the system heap.

Unfortunately, manually copying a routine into the system heap is difficult in Pascal. The easiest way to install code into the system heap is to place the code into a separate stand-alone code resource in your application's resource fork. You should set the system heap bit and the locked bit of the code resource's attributes. Then, when you need to use the code, you must load the resource from the resource file and cast the resource handle's master pointer into a procedure pointer (a variable of type `ProcPtr`), as follows:

```
myProcHandle := GetResource(kProcType, kProcID);
IF myProcHandle <> NIL THEN
    myTaskPtr^.vblAddr := ProcPtr(myProcHandle^);
```

Because the resource is locked in memory, you don't have to worry about creating a dangling pointer when you dereference a handle to the resource. If you want the code to remain in the system heap after the user quits your application, you can call the Resource Manager procedure `DetachResource` so that closing your application's resource fork does not destroy the resource data. Note, however, that if you do so and your application crashes, the code still remains in the system heap.

Once you have loaded a code resource into memory and created a `ProcPtr` that references the entry point of the code resource, you can use that `ProcPtr` just as you can use any such variable. For example, you could assign the value of the variable to the `vblAddr` field of a vertical retrace task record (as shown just above). If you are programming in assembly language, you can then call the code directly. To call the routine from a high-level language such as Pascal, you'll need to use some inline assembly-language code. Listing 6-4 defines a routine that you can use to execute a procedure by address.

Listing 6-4 Calling a procedure by address

```

PROCEDURE CallByAddress (aRoutine: ProcPtr);
    INLINE    $205F,          {MOVE.L (SP)+,A0}
              $4ED0;          {JMP (A0)}

```

Allocating Memory at Startup Time

If you are implementing a system extension, you might need to allocate memory at startup time. As explained in the previous section, an ideal place to allocate such memory is in the system heap. To allocate memory in the system heap under system software version 7.0 and later, you merely need to call the appropriate Memory Manager routines, and the system heap expands dynamically to meet your request. In earlier versions of system software, you must use a 'sysz' resource to indicate how much the Operating System should increase the size of the system zone.

Alternatively, however, you can allocate blocks in high memory. The global variable `BufPtr` always references the highest byte in memory that might become part of an application partition. You can lower the value of `BufPtr` and then use the memory between the old and new values of `BufPtr`.

Note

In general, if you are implementing a system extension, you should allocate memory in the system heap instead of high memory. In this way, you avoid the problems associated with lowering the value of `BufPtr` too far (described in the following paragraphs) and ensure that the extension is not paged out if virtual memory is operating. ♦

Lowering the value of `BufPtr` too far can be dangerous for several reasons. In 128K ROM Macintosh computers running system software version 4.1, you must avoid lowering the value of `BufPtr` so that it points in the system startup blocks. The highest byte of these blocks can always be found relative to the global variable `MemTop`, at `MemTop DIV 2 + 1024`.

In later versions of the Macintosh system software, the system startup blocks were no longer barriers to `BufPtr`, but new barriers arose, including Macintosh IIci video storage, for example. To maintain compatibility with extensions that rely on the ability to lower `BufPtr` relative to `MemTop`, the system software simply adjusts `MemTop` so that the formula still holds. Thus, at startup, the `MemTop` global variable currently does not reference any memory location in particular. Instead, it holds a value that guarantees that the formula allowing you to lower `BufPtr` as low as `MemTop DIV 2 + 1024` but no further still holds.

Beginning in system software version 7.0, the Operating System can detect excessive lowering of `BufPtr`, but only after the fact. When the Operating System does detect that the value of `BufPtr` has fallen too low, it generates an out-of-memory system error.

▲ **WARNING**

Although the above formula has been true since system software version 4.1, a bug in the Macintosh IIci and later ROMs made it invalid in certain versions of system software 6.x. ▲

Because there is no calling interface for lowering `BufPtr`, you must do it manually, by changing the value of the system variable, as explained in “Reading and Writing System Global Variables” on page 6-320. To obtain the value of the `MemTop` global variable, you can use the `TopMem` function.

Creating Heap Zones

You can create heap zones as subzones of your application heap zone or (in rare instances) either in space reserved for the application global variables or on the stack. You can also create heap zones in a block of temporary memory or within the system heap zone. This section describes how to create new heap zones by calling the `InitZone` procedure.

Note

Most applications do not need to create heap zones. ♦

To create a new heap zone in the application heap, you must allocate nonrelocatable blocks in your application heap to hold new subzones of the application heap. In addition to being able to create subzones of the application zone, you can create subzones of any other zone to which you have access, including a zone that is itself a subzone of another zone.

You create a heap zone by calling the `InitZone` procedure, which takes four parameters. The first parameter specifies a grow-zone function for the new zone, or `NIL` if you do not want the zone to have a grow-zone function. The second parameter specifies the number of new master pointers that you want each block of master pointers in the zone to contain. The `InitZone` procedure allocates one such block to start with, and you can allocate more by calling the `MoreMasters` procedure. The third and fourth parameters specify, respectively, the first byte beyond the end of the new zone and the first byte of the zone.

When initializing a zone with the `InitZone` procedure, make sure that you are subdividing the current zone. When `InitZone` returns, the new zone becomes current. Thus, if you subdivide the application zone into several subzones, you must call `SetZone(ApplicationZone)` before you create the second and each of the subsequent subzones. Listing 6-5 shows a technique for creating a single subzone of the original application zone, assuming that the application zone is the current zone. The technique for subdividing subzones is similar.

Listing 6-5 Creating a subzone of the original application heap zone

```

FUNCTION CreateSubZone: THz;
CONST
    kZoneSize = 10240;           {10K zone}
    kNumMasterPointers = 16;     {num of master ptrs for new zone}
VAR
    start:    Ptr;               {first byte in zone}
    limit:    Ptr;               {first byte beyond zone}
BEGIN
    start := NewPtr(kZoneSize);  {allocate storage for zone}
    IF MemError <> noErr THEN
        BEGIN
            limit := Ptr(ORD4(start) + kZoneSize);
                                   {compute byte beyond end of zone}
            InitZone(NIL, kNumMasterPointers, limit, start);
                                   {initialize zone header, trailer}
        END;
    CreateSubZone := THz(start); {cast storage to a zone pointer}
END;

```

To create a subzone in the system heap zone, you can call `SetZone(SystemZone)` at the beginning of the procedure in Listing 6-5. You might find this technique useful if you are implementing a system extension but want to manage your extension's memory much as you manage memory in an application. Instead of simply allocating blocks in the system heap, you can make your zone current whenever your extension is executed. Then, you can call regular Memory Manager routines to allocate memory in your subzone of the system heap, and you can compact and purge your subzone without compacting and purging the entire system heap zone.

When you allocate memory for a subzone, you must allocate that memory in a nonrelocatable block (as in Listing 6-5) or in a locked relocatable block. If you create a subzone within an unlocked relocatable block, the Memory Manager might move your entire subzone during memory operations in the zone containing your subzone. If so, any references to nonrelocatable blocks that you allocated in the subzone would become invalid. Even handles to relocatable blocks in the subzone would no longer be valid, because the Memory Manager does not update the handles' master pointers correctly. This happens because the Memory Manager views a subzone of another zone as a single block. If that subzone is a relocatable block, the Memory Manager updates only that block's master pointer when moving it, and does not update the block's contents (that is, the blocks allocated within the subzone).

If you use a block of temporary memory as a heap zone, you must lock the temporary memory immediately after allocating it. Then, you can pass to `InitZone` a dereferenced copy of a handle to the temporary memory. If you find (after a call to the `Gestalt` function) that temporary memory handles are not real, then you must dispose of the new zone before any calls to `GetNextEvent` or `WaitNextEvent`. You must dispose of the

new zone because you cannot lock a handle to temporary memory across event calls if the handle is not real.

Once you have created a subzone as a nonrelocatable block or a locked relocatable block, you can allocate both relocatable and nonrelocatable blocks within it. Although the Memory Manager can move such relocatable blocks only within the subzone, it correctly updates those blocks' master pointers, which are also in the subzone.

Installing a Purge-Warning Procedure

You can define a **purge-warning procedure** that the Memory Manager calls whenever it is about to purge a block from your application heap. You can use this procedure to save the data in the block, if necessary, or to perform other processing in response to this notification.

Note

Most applications don't need to install a purge-warning procedure. This capability is provided primarily for applications that require greater control over their heap. Examples are applications that maintain purgeable handles containing important data and applications that for any other reason need notification when a block is about to be purged. ♦

When your purge-warning procedure is called, the Memory Manager passes it a handle to the block about to be purged. In your procedure, you can test the handle to determine whether it contains data that needs to be saved; if so, you can save the data (possibly by writing it to some open file). Listing 6-6 defines a very simple purge-warning procedure.

Listing 6-6 A purge-warning procedure

```
PROCEDURE MyPurgeProc (h: Handle);
VAR
    theA5:    LongInt;           {value of A5 when procedure is called}
BEGIN
    theA5 := SetCurrentA5;       {remember current value of A5; install ours}
    IF BAND(HGetState(h), $20) = 0 THEN
        BEGIN                   {if the handle isn't a resource handle}
            IF InSaveList(h) THEN
                WriteData(h);    {save the data in the block}
            END;
        theA5 := SetA5(theA5);   {restore previous value of A5}
    END;
```

The MyPurgeProc procedure defined in Listing 6-6 inspects the handle's properties (using HGetState) to see whether its resource bit is clear. If so, the procedure next determines whether the handle is contained in an application-maintained list of handles whose data should be saved before purging. If the handle is in that list, the purge-warning procedure writes its data to disk. (The file into which the data is written

Memory Manager

should already be open at the time the procedure is called, because opening a file might cause memory to move.)

Note that `MyPurgeProc` sets up the A5 register with the application's A5 value upon entry and restores it to its previous value before exiting. This is necessary because you cannot rely on the A5 register within a purge-warning procedure.

▲ **WARNING**

Because of the optimizations performed by some compilers, the actual work of the purge-warning procedure and the setting and restoring of the A5 register might have to be placed in separate procedures. See the chapter “Vertical Retrace Manager” in *Inside Macintosh: Processes* for an illustration of how you can do this. ▲

To install a purge-warning procedure, you need to install the address of the procedure into the `purgeProc` field of your application's heap zone header. Listing 6-7 illustrates one way to do this.

Listing 6-7 Installing a purge-warning procedure

```
PROCEDURE InstallPurgeProc;
VAR
    myZone: THz;
BEGIN
    myZone := GetZone;           {find the current zone header}
    gPrevProc := myZone^.purgeProc; {remember previous procedure}
    myZone^.purgeProc := @MyPurgeProc; {install new procedure}
END;
```

The `InstallPurgeProc` procedure defined in Listing 6-7 first obtains the address of the current heap zone by calling the `GetZone` function. Then it saves the address of any existing purge-warning procedure in the global variable `gPrevProc`. Finally, `InstallPurgeProc` installs the new procedure by putting its address directly into the `purgeProc` field of the zone header. (For more information on zone headers, see “Heap Zones” on page 6-332.)

Keep in mind that the Memory Manager calls your purge-warning procedure each time it decides to purge any purgeable block, and it might call your procedure far more often than you would expect. Your purge-warning procedure might be passed handles not only to blocks that you explicitly mark as purgeable (by calling `HPurge`), but also to resources whose purgeable attribute is set. (In general, applications don't need to take any action on handles that belong to the Resource Manager.) Because of the potentially large number of times your purge-warning procedure might be called, it should be able to determine quickly whether a handle that is about to be purged needs additional processing.

Remember that a purge-warning procedure is called during the execution of some Memory Manager routine. As a result, your procedure cannot cause memory to be

moved or purged. In addition, it should not dispose of the handle it is passed or change the purge status of the handle. See “Purge-Warning Procedures” on page 6-403 for a complete description of the limitations on purge-warning procedures.

▲ **WARNING**

If your application calls the Resource Manager procedure `SetResPurge` with the parameter `TRUE` (to have the Resource Manager automatically save any modified resources that are about to be purged), you should avoid using a purge-warning procedure. This is because the Resource Manager installs its own purge-warning procedure when you call `SetResPurge` in this way. If you must install your own purge-warning procedure, you should remove your procedure, call `SetResPurge`, then reinstall your procedure as shown in Listing 6-7. You then need to make sure that your procedure calls the Resource Manager’s purge-warning procedure (which is saved in the global variable `gPrevProc`) before exiting. Most applications do not need to call `SetResPurge` at all. ▲

If your application does call `SetResPurge(TRUE)`, you should use the version of `MyPurgeProc` defined in Listing 6-8. It is just like the version defined in Listing 6-6 except that it calls the Resource Manager’s purge-warning procedure before exiting.

Listing 6-8 A purge-warning procedure that calls the Resource Manager’s procedure

```
PROCEDURE MyPurgeProc (h: Handle);
VAR
    theA5:    LongInt;                {value of A5 when procedure is called}
BEGIN
    theA5 := SetCurrentA5;            {remember current value of A5; install ours}
    IF BAND(HGetState(h), $20) = 0 THEN
        BEGIN                        {if the handle isn't a resource handle}
            IF InSaveList(h) THEN
                WriteData(h);        {save the data in the block}
            END
        ELSE IF gPrevProc <> NIL THEN
            CallByAddress(gPrevProc);
        theA5 := SetA5(theA5);        {restore previous value of A5}
    END;
```

See Listing 6-4 on page 6-326 for a definition of the procedure `CallByAddress`.

Organization of Memory

This section describes the organization of heap zones and block headers. In general, you do not need to know how the Memory Manager organizes heap zones or block headers if

Memory Manager

your application simply allocates and releases blocks of memory. The information described in this section is used by the Memory Manager for its own purposes. Developers of some specialized applications and utilities might, however, need to know exactly how zones and block headers are organized. This information is also sometimes useful for debugging.

▲ **WARNING**

This section is provided primarily for informational purposes. The organization and size of heap zones and block headers is subject to change in future system software versions. ▲

Heap Zones

Except for temporary memory blocks, all relocatable and nonrelocatable blocks exist within heap zones. A heap zone consists of a zone header, a zone trailer block, and usable bytes in between. The header contains all of the information the Memory Manager needs about that heap zone; the trailer is just a minimum-sized free block placed as a marker at the end of the zone.

In Pascal, a heap zone is defined as a **zone record** of type `Zone`. The zone record contains all of the fields of the zone header. A heap zone is always referred to with a **zone pointer** of data type `THz`.

▲ **WARNING**

The fields of the zone header are for the Memory Manager's own internal use. You can examine the contents of the zone's fields, but in general it doesn't make sense for your application to try to change them. The only fields of the zone record that you can safely modify directly are the `moreMast` and `purgeProc` fields. ▲

```

TYPE Zone =
RECORD
    bkLim:      Ptr;           {first usable byte after zone}
    purgePtr:   Ptr;           {used internally}
    hFstFree:   Ptr;           {first free master pointer}
    zcbFree:    LongInt;       {number of free bytes in zone}
    gzProc:     ProcPtr;       {grow-zone function}
    moreMast:   Integer;       {num. of master ptrs to allocate}
    flags:      Integer;       {used internally}
    cntRel:     Integer;       {reserved}
    maxRel:     Integer;       {reserved}
    cntNRel:    Integer;       {reserved}
    maxNRel:    Integer;       {reserved}
    cntEmpty:   Integer;       {reserved}
    cntHandles: Integer;       {reserved}
    minCBFree:  LongInt;       {reserved}
    purgeProc:  ProcPtr;       {purge-warning procedure}

```

Memory Manager

```

    sparePtr:    Ptr;           {used internally}
    allocPtr:    Ptr;           {used internally}
    heapData:    Integer;       {first usable byte in zone}
END;
THz = ^Zone;           {zone pointer}

```

Field descriptions

bkLim	A pointer to the byte <i>following</i> the last byte of usable space in the zone.
purgePtr	Used internally.
hFstFree	A pointer to the first free master pointer in the zone. All master pointers that are allocated but not currently in use are linked together into a list. The hFstFree field references the head node of this list. The Memory Manager updates this list every time it allocates a new relocatable block or releases one, so that the list contains all unused master pointers. If the Memory Manager needs a new master pointer but this field is set to NIL, it allocates a new nonrelocatable block of master pointers. You can check the value of this field to see whether allocating a relocatable block would cause a new block of master pointers to be allocated.
zcbFree	The number of free bytes remaining in the zone. As blocks are allocated and released, the Memory Manager adjusts this field accordingly. You can use the FreeMem function to determine the value of this field for the current heap zone.
gzProc	A pointer to a grow-zone function that system software uses to maintain control over the heap. The system's grow-zone function subsequently calls the grow-zone function you specify for your heap, if any. You can change a heap zone's grow-zone function at any time but should do so only by calling the InitZone or SetGrowZone procedures. Note that in current versions of system software, this field does not contain a pointer to the grow-zone function that your application defines.
moreMast	The number of master pointers the Memory Manager should allocate at a time. The Memory Manager allocates this many automatically when a heap zone is initialized. By default, master pointers are allocated 32 at a time for the system heap zone and 64 at a time for the application heap zone, but this might change in future versions of system software.
flags	Used internally.
cntRel	Reserved.
maxRel	Reserved.
cntNRel	Reserved.
maxNRel	Reserved.
cntEmpty	Reserved.
cntHandles	Reserved.
minCBFree	Reserved.

Memory Manager

<code>purgeProc</code>	A pointer to the zone's purge-warning procedure, or <code>NIL</code> if there is none. The Memory Manager calls this procedure before it purges a block from the zone. Note that whenever you call the Resource Manager procedure <code>SetResPurge</code> with the parameter set to <code>TRUE</code> , the Resource Manager installs its own purge-warning procedure, overriding any purge-warning procedure you have specified here.
<code>sparePtr</code>	Used internally.
<code>allocPtr</code>	Used internally.
<code>heapData</code>	A dummy field marking the beginning of the zone's usable memory space. The integer in this field has no significance in itself; it is just the first 2 bytes in the block header of the first block in the zone. For example, if <code>myZone</code> is a zone pointer, then <code>@(myZone^.heapData)</code> is the address of the first usable byte in the zone, and <code>myZone^.bkLim</code> is a pointer to the byte following the last usable byte in the zone.

The structure of a heap zone is the same in both 24-bit and 32-bit addressing modes. The use of several of the fields that are reserved or used internally, however, may differ in 24-bit and 32-bit heap zones.

Block Headers

Every block in a heap zone, whether allocated or free, has a **block header** that the Memory Manager uses to find its way around in the zone. Block headers are completely transparent to your application. All pointers and handles to allocated blocks reference the beginning of the block's logical contents, following the end of the header. Similarly, whenever you use a variable of type `Size`, that variable refers to the number of bytes in the block's logical contents, not including the block header. That size is known as the block's **logical size**, as opposed to its **physical size**, the number of bytes it actually occupies in memory, including the header and any unused bytes at the end of the block.

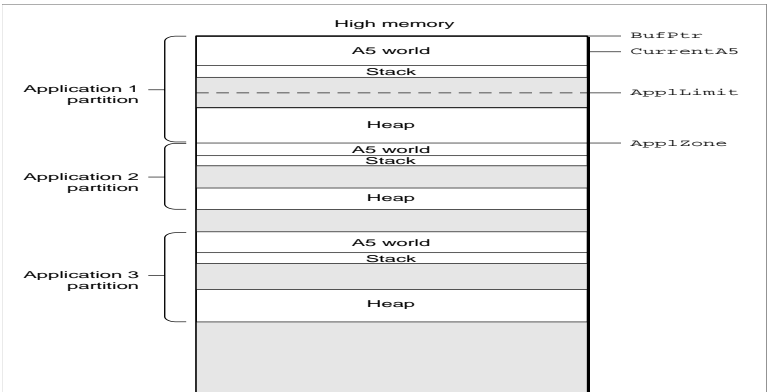
There are two reasons that a block might contain such unused bytes:

- The Memory Manager allocates space only in even numbers of bytes. (This practice guarantees that both the contents and the address of a master pointer are even.) If a block's logical size is odd, an extra, unused byte is added at the end to make the physical size an even number. On computers containing the MC68020, MC68030, or MC68040 microprocessor, blocks are padded to 4-byte boundaries.
- The minimum number of bytes in a block is 12. This minimum applies to all blocks, free as well as allocated. If allocating the required number of bytes from a free block would leave a fragment of fewer than 12 free bytes, the leftover bytes are included unused at the end of the newly allocated block instead of being returned to free storage.

There is no Pascal record type defining the structure of block headers because you shouldn't normally need to access them directly. In addition, the structure of a block header depends on whether the block is located in a 24-bit or 32-bit zone.

In a 24-bit zone, a block header consists of 8 bytes, which together make up two long words, as shown in Figure 6-1.

Figure 6-1 A block header in a 24-bit zone



In the first long word, the low-order 3 bytes contain the block's physical size in bytes. Adding this number to the block's address gives the address of the next block in the zone. The first byte of the block header is a **tag byte** that provides other information on the block. The bits in the tag byte have these meanings:

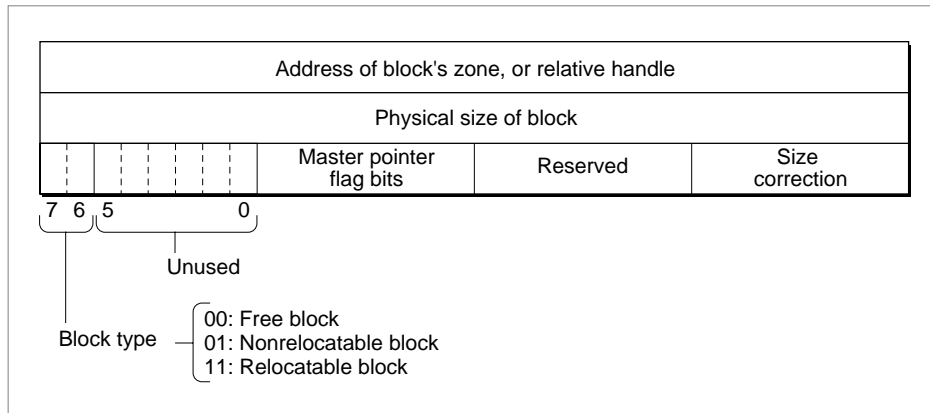
Bit	Meaning
0-3	The block's size correction
4-5	Reserved
6-7	The block type

In the tag byte, the high-order 2 bits determine whether a block is free (binary 00), relocatable (binary 10), or nonrelocatable (binary 01). The low-order 4 bits contain a block's **size correction**, the number of unused bytes at the end of the block, beyond the end of the block's contents. This correction is equal to the difference between the block's logical and physical sizes, excluding the 8 bytes of overhead for the block header, as in the following formula:

$$\text{physicalSize} = \text{logicalSize} + \text{sizeCorrection} + 8$$

The contents of the second long word (4 bytes) in the 24-bit block header depend on the type of block. For relocatable blocks, the second long word contains the block's **relative handle**: a pointer to the block's master pointer, expressed as an offset relative to the start of the heap zone rather than as an absolute memory address. Adding the relative handle to the zone pointer produces a true handle for this block. For nonrelocatable blocks, the second long word of the header is just a pointer to the block's zone. For free blocks, the contents of these 4 bytes are undefined.

In a 32-bit zone, a block header consists of 12 bytes, which together make up three long words, as shown in Figure 6-2.

Figure 6-2 A block header in a 32-bit zone

The first byte of the block header is a tag byte that indicates the type of the block. The bits in the tag byte have these meanings:

Bit	Meaning
0–5	Reserved
6–7	The block type

In the tag byte, the high-order 2 bits determine whether a block is free (binary 00), relocatable (binary 10), or nonrelocatable (binary 01).

The second byte in the block header contains the master pointer flag bits, if the block is a relocatable block. Otherwise, this byte is undefined. The bits in this byte have these meanings:

Bit	Meaning
0–4	Reserved
5	If set, block contains resource data
6	If set, block is purgeable
7	If set, block is locked

The low-order byte of the high-order long word contains the block's size correction. This correction is equal to the difference between the block's logical and physical sizes, excluding the 12 bytes of overhead for the block header, as follows:

$$\text{physicalSize} = \text{logicalSize} + \text{sizeCorrection} + 12$$

The second long word in the 32-bit block header contains the block's physical size, and the third long word contains the block's relative handle. These fields have the same meaning as the corresponding fields in the 24-bit block header.

Memory Manager Reference

This section describes the data types and routines provided by the Memory Manager. It describes the general-purpose data types the Memory Manager defines and all routines that relate to manipulating blocks of memory or managing memory in the application heap zone. This section also describes the data structures and routines that allow your application to allocate temporary memory and to use multiple heap zones.

Data Types

This section discusses the general-purpose data types defined by the Memory Manager. Most of these types are used throughout the system software.

The Memory Manager uses pointers and handles to reference nonrelocatable and relocatable blocks, respectively. The data types `Ptr` and `Handle` define pointers and handles as follows:

```

TYPE
    SignedByte    = -128..127;
    Byte          = 0..255;
    Ptr           = ^SignedByte;
    Handle        = ^Ptr;
  
```

The `SignedByte` type stands for an arbitrary byte in memory, just to give `Ptr` and `Handle` something to point to. The `Byte` type is an alternative definition that treats byte-length data as an unsigned rather than a signed quantity.

Many other data types also use the concept of pointers and handles. For example, the Macintosh system software stores strings in arrays of up to 255 characters, with the first byte of the array storing the length of the string. Some Toolbox routines allow you to pass such a string directly; others require that you pass a pointer or handle to a string. The following type definitions define character strings:

```

TYPE
    Str255        = STRING[255];
    StringPtr     = ^Str255;
    StringHandle  = ^StringPtr;
  
```

Some Toolbox routines allow you to execute code after a certain amount of time elapses or after a certain condition is met. Any such routine requires you to pass the address of the routine containing the code to be executed so that it knows what routine to call when the time has elapsed or the condition has been met. You use the data type `ProcPtr` to define a pointer to a procedure or function.

```

TYPE ProcPtr = Ptr;
  
```

Memory Manager

For example, after the declarations

```
VAR
    aProcPtr: ProcPtr;

PROCEDURE MyProc;
BEGIN
    ...
END;
```

you can make `aProcPtr` reference the `MyProc` procedure by using the `@` operator, as follows:

```
aProcPtr := @MyProc;
```

With the `@` operator, you can assign procedures and functions to variables of type `ProcPtr`, embed them in data structures, and pass them as arguments to other routines. Notice, however, that the data type `ProcPtr` technically points to an arbitrary byte, not an actual routine. As a result, there's no direct way in Pascal to access the underlying routine via this pointer in order to call it. (See Listing 6-4 on page 6-326 for some assembly-language code you can use to do so.) The routines in the Operating System and Toolbox, which are written in assembly language, can however, call routines designated by pointers of type `ProcPtr`.

Note

You can't use the `@` operator to reference procedures or functions whose declarations are nested within other routines. ♦

The Memory Manager uses the `Size` data type to refer to the size, in bytes, of memory blocks. For example, when specifying how large a relocatable block you want to allocate, you pass a parameter of type `Size`. The `Size` data type is also defined as a long integer.

```
TYPE Size      = LongInt;
```

Memory Manager Routines

This section describes the routines provided by the Memory Manager. You can use these routines to set up your application's partition, allocate and dispose of relocatable and nonrelocatable blocks, manipulate those blocks, assess the availability of memory in your application's heap, free memory from the heap, and install a grow-zone function for your heap. The Memory Manager also provides routines that allow you to allocate temporary memory and manipulate heap zones.

Note

The result codes listed for Memory Manager routines are usually not directly returned to your application. You need to call the `MemError` function (or, from assembly language, inspect the `MemErr` global variable) to get a routine's result code. ♦

You cannot call most Memory Manager routines at interrupt time for several reasons. You cannot allocate memory at interrupt time because the Memory Manager might already be handling a memory-allocation request and the heap might be in an inconsistent state. More generally, you cannot call at interrupt time any Memory Manager routine that returns its result code via the `MemError` function, even if that routine doesn't allocate or move memory. Resetting the `MemErr` global variable at interrupt time can lead to unexpected results if the interrupted code depends on the value of `MemErr`. Note that Memory Manager routines like `HLock` return their results via `MemError` and therefore should not be called in interrupt code.

Setting Up the Application Heap

The Operating System automatically initializes your application's heap when your application is launched. To help prevent heap fragmentation, you should call the procedures in this section before you allocate any blocks of memory in your heap.

Use the `MaxApplZone` procedure to extend the application heap zone to the application heap limit so that the Memory Manager does not do so gradually as memory requests require. Use the `MoreMasters` procedure to preallocate enough blocks of master pointers so that the Memory Manager never needs to allocate new master pointer blocks for you.

MaxApplZone

To help ensure that you can use as much of the application heap zone as possible, call the `MaxApplZone` procedure. Call this once near the beginning of your program, after you have expanded your stack.

```
PROCEDURE MaxApplZone ;
```

DESCRIPTION

The `MaxApplZone` procedure expands the application heap zone to the application heap limit. If you do not call `MaxApplZone`, the application heap zone grows as necessary to fulfill memory requests. The `MaxApplZone` procedure does not purge any blocks currently in the zone. If the zone already extends to the limit, `MaxApplZone` does nothing.

It is a good idea to call `MaxApplZone` once at the beginning of your program if you intend to maintain an effectively partitioned heap. If you do not call `MaxApplZone` and

Memory Manager

then call `MoveHHi` to move relocatable blocks to the top of the heap zone before locking them, the heap zone could later grow beyond these locked blocks to fulfill a memory request. If the Memory Manager were to allocate a nonrelocatable block in this new space, your heap would be fragmented.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `MaxApplZone` are

Registers on exit

D0 Result code

RESULT CODES

`noErr` 0 No error

MoreMasters

Call the `MoreMasters` procedure several times at the beginning of your program to prevent the Memory Manager from running out of master pointers in the middle of application execution. If it does run out, it allocates more, possibly causing heap fragmentation.

```
PROCEDURE MoreMasters;
```

DESCRIPTION

The `MoreMasters` procedure allocates another block of master pointers in the current heap zone. In the application heap, a block of master pointers consists of 64 master pointers, and in the system heap, a block consists of 32 master pointers. (These values, however, might change in future versions of system software.) When you initialize additional heap zones, you can specify the number of master pointers you want to have in a block of master pointers.

The Memory Manager automatically calls `MoreMasters` once for every new heap zone, including the application heap zone.

You should call `MoreMasters` at the beginning of your program enough times to ensure that the Memory Manager never needs to call it for you. For example, if your application never allocates more than 300 relocatable blocks in its heap zone, then five calls to the `MoreMasters` should be enough. It's better to call `MoreMasters` too many times than too few. For instance, if your application usually allocates about 100 relocatable blocks but might allocate 1000 in a particularly busy session, call `MoreMasters` enough times at the beginning of the program to accommodate times of greater memory use.

If you are forced to call `MoreMasters` so many times that it causes a significant slowdown, you could change the `moreMast` field of the zone header to the total number

of master pointers you need and then call `MoreMasters` just once. Afterward, be sure to restore the `moreMast` field to its original value.

SPECIAL CONSIDERATIONS

Because `MoreMasters` allocates memory, you should not call it at interrupt time.

The calls to `MoreMasters` at the beginning of your application should be in the main code segment of your application or in a segment that the main segment never unloads.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `MoreMasters` are

Registers on exit

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory

SEE ALSO

If you initialize a new zone, you can specify the number of master pointers that a master pointer block should contain. See the description of the `InitZone` procedure on page 6-399 for details.

Allocating and Releasing Relocatable Blocks of Memory

You can use the `NewHandle` function to allocate a relocatable block of memory, or the `NewEmptyHandle` function to allocate handles for which you do not yet need blocks of memory. If you want to allocate new blocks of memory in the system heap or with their bits precleared to 0, you can use the functions `NewHandleSys`, `NewHandleClear`, and `NewHandleSysClear`.

▲ WARNING

You should not call any of these memory-allocation routines at interrupt time. ▲

You can use the `DisposeHandle` procedure to free relocatable blocks of memory you have allocated.

NewHandle

You can use the `NewHandle` function to allocate a relocatable memory block of a specified size.

```
FUNCTION NewHandle (logicalSize: Size): Handle;
```

`logicalSize`

The requested size (in bytes) of the relocatable block.

DESCRIPTION

The `NewHandle` function attempts to allocate a new relocatable block in the current heap zone with a logical size of `logicalSize` bytes and then return a handle to the block. The new block is unlocked and unpurgeable. If `NewHandle` cannot allocate a block of the requested size, it returns `NIL`.

▲ WARNING

Do not try to manufacture your own handles without this function by simply assigning the address of a variable of type `Ptr` to a variable of type `Handle`. The resulting “fake handle” would not reference a relocatable block and could cause a system crash. ▲

The `NewHandle` function pursues all available avenues to create a block of the requested size, including compacting the heap zone, increasing its size, and purging blocks from it. If all of these techniques fail and the heap zone has a grow-zone function installed, `NewHandle` calls the function. Then `NewHandle` tries again to free the necessary amount of memory, once more compacting and purging the heap zone if necessary. If memory still cannot be allocated, `NewHandle` calls the grow-zone function again, unless that function had returned 0, in which case `NewHandle` gives up and returns `NIL`.

SPECIAL CONSIDERATIONS

Because `NewHandle` allocates memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `NewHandle` are

Registers on entry

A0 Number of logical bytes requested

Registers on exit

A0 Address of the new block's master pointer or
 `NIL`

D0 Result code

Memory Manager

You can specify that the `NewHandle` function apply to the system heap zone instead of the current zone by setting bit 10 of the routine trap word. In most development systems, you can do this by supplying the word `SYS` as the second argument to the routine macro, as follows:

```
_NewHandle ,SYS
```

If you want to clear the bytes of a block of memory to 0 when you allocate it with the `NewHandle` function, set bit 9 of the routine trap word. You can usually do this by supplying the word `CLEAR` as the second argument to the routine macro, as follows:

```
_NewHandle ,CLEAR
```

You can combine `SYS` and `CLEAR` in the same macro call, but `SYS` must come first.

```
_NewHandle ,SYS ,CLEAR
```

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory in heap zone

SEE ALSO

If you allocate a relocatable block that you plan to lock for long periods of time, you can prevent heap fragmentation by allocating the block as low as possible in the heap zone. To do this, see the description of the `ReserveMem` procedure on page 6-368.

If you plan to lock a relocatable block for short periods of time, you might want to move it to the top of the heap zone to prevent heap fragmentation. For more information, see the description of the `MoveHHI` procedure on page 6-369.

NewHandleSys

You can use the `NewHandleSys` function to allocate a relocatable block of memory of a specified size in the system heap.

```
FUNCTION NewHandleSys (logicalSize: Size): Handle;
```

`logicalSize`

The requested size (in bytes) of the relocatable block.

DESCRIPTION

The `NewHandleSys` function works much as the `NewHandle` function does, but attempts to allocate the requested block in the system heap zone instead of in the current heap zone. If it cannot, it returns `NIL`.

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory in heap zone

NewHandleClear

You can use the `NewHandleClear` function to allocate prezeroed memory in a relocatable block of a specified size.

```
FUNCTION NewHandleClear (logicalSize: Size): Handle;
```

`logicalSize`

The requested size (in bytes) of the relocatable block. The `NewHandleClear` function sets each of these bytes to 0.

DESCRIPTION

The `NewHandleClear` function works much as the `NewHandle` function does but sets all bytes in the new block to 0 instead of leaving the contents of the block undefined.

Currently, `NewHandleClear` clears the block one byte at a time. For a large block, it might be faster to clear the block manually a long word at a time.

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory in heap zone

NewHandleSysClear

You can use the `NewHandleSysClear` function to allocate, in the system heap, prezeroed memory in a relocatable block of a specified size.

```
FUNCTION NewHandleSysClear (logicalSize: Size): Handle;
```

`logicalSize`

The requested size (in bytes) of the relocatable block. The `NewHandleSysClear` function sets each of these bytes to 0.

DESCRIPTION

The `NewHandleSysClear` function works much as the `NewHandleClear` function does, but attempts to allocate the requested block in the system heap zone instead of in the current heap zone. `NewHandleSysClear` sets all bytes in the new block to 0 instead of leaving the contents of the block undefined.

RESULT CODES

noErr	0	No error
memFullErr	-108	Not enough memory in heap zone

NewEmptyHandle

If you want to initialize a handle but not allocate any space for it, use the `NewEmptyHandle` function. The Resource Manager uses this function extensively, but you probably won't need to use it.

```
FUNCTION NewEmptyHandle: Handle;
```

DESCRIPTION

The `NewEmptyHandle` function initializes a new handle by allocating a master pointer for it, but it does not allocate any memory for the handle to control. `NewEmptyHandle` sets the handle's master pointer to `NIL`.

SPECIAL CONSIDERATIONS

Because `NewEmptyHandle` might need to call the `MoreMasters` procedure to allocate new master pointers, it might allocate memory. Thus, you should not call `NewEmptyHandle` at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `NewEmptyHandle` are

Registers on exit

A0	Address of the new block's master pointer
D0	Result code

You can specify that the `NewEmptyHandle` function apply to the system heap zone instead of the current zone. To do so, set bit 10 of the routine trap word. In most development systems, you can do this by supplying the word `SYS` as the second argument to the routine macro, as follows:

```
_NewEmptyHandle ,SYS
```

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory

SEE ALSO

When you want to allocate memory for the empty handle, use the `ReallocateHandle` procedure, described on page 6-365.

NewEmptyHandleSys

If you want to initialize a handle in the system heap but not allocate any space for it, use the `NewEmptyHandleSys` function. The Resource Manager uses this function extensively, but you probably won't need to use it.

```
FUNCTION NewEmptyHandleSys: Handle;
```

DESCRIPTION

The `NewEmptyHandleSys` function initializes a new handle in the system heap by allocating a master pointer for it, but it does not allocate any memory for the handle to control. `NewEmptyHandleSys` sets the handle's master pointer to `NIL`.

SPECIAL CONSIDERATIONS

Because `NewEmptyHandleSys` might need to call the `MoreMasters` procedure to allocate new master pointers, it might allocate memory. Thus, you should not call `NewEmptyHandleSys` at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `NewEmptyHandleSys` are

Registers on exit

A0	Address of the new block's master pointer
D0	Result code

RESULT CODES

noErr	0	No error
memFullErr	-108	Not enough memory

SEE ALSO

When you want to allocate memory for the empty handle, use the `ReallocateHandle` procedure, described on page 6-365.

DisposeHandle

When you are completely done with a relocatable block, call the `DisposeHandle` procedure to free it and its master pointer for other uses.

```
PROCEDURE DisposeHandle (h: Handle);
```

h A handle to a relocatable block.

DESCRIPTION

The `DisposeHandle` procedure releases the memory occupied by the relocatable block whose handle is `h`. It also frees the handle's master pointer for other uses.

▲ WARNING

After a call to `DisposeHandle`, all handles to the released block become invalid and should not be used again. Any subsequent calls to `DisposeHandle` using an invalid handle might damage the master pointer list. ▲

Do not use `DisposeHandle` to dispose of a handle obtained from the Resource Manager (for example, by a previous call to `GetResource`); use `ReleaseResource` instead. If, however, you have called `DetachResource` on a resource handle, you should dispose of the storage by calling `DisposeHandle`.

SPECIAL CONSIDERATIONS

Because `DisposeHandle` purges memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `DisposeHandle` are

Registers on entry

A0 Handle to the relocatable block to be disposed of

Registers on exit

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>memWZErr</code>	-111	Attempt to operate on a free block

Allocating and Releasing Nonrelocatable Blocks of Memory

You can use the `NewPtr` function to allocate a nonrelocatable block of memory. If you want to allocate new blocks of memory in the system heap or with their bits precleared to 0, you can use the `NewPtrSys`, `NewPtrClear`, and `NewPtrSysClear` functions.

▲ WARNING

You should not call any of these memory-allocation routines at interrupt time. ▲

You can use the `DisposePtr` procedure to free nonrelocatable blocks of memory you have allocated.

NewPtr

You can use the `NewPtr` function to allocate a nonrelocatable block of memory of a specified size.

```
FUNCTION NewPtr (logicalSize: Size): Ptr;
```

`logicalSize`

The requested size (in bytes) of the nonrelocatable block.

DESCRIPTION

The `NewPtr` function attempts to allocate, in the current heap zone, a nonrelocatable block with a logical size of `logicalSize` bytes and then return a pointer to the block. If the requested number of bytes cannot be allocated, `NewPtr` returns `NIL`.

The `NewPtr` function attempts to reserve space as low in the heap zone as possible for the new block. If it is able to reserve the requested amount of space, `NewPtr` allocates the nonrelocatable block in the gap `ReserveMem` creates. Otherwise, `NewPtr` returns `NIL` and generates a `memFullErr` error.

SPECIAL CONSIDERATIONS

Because `NewPtr` allocates memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `NewPtr` are

Registers on entry

A0 Number of logical bytes requested

Registers on exit

A0 Address of the new block or
 NIL

D0 Result code

You can specify that the `NewPtr` function apply to the system heap zone instead of the current zone. To do so, set bit 10 of the routine trap word. In most development systems, you can do this by supplying the word `SYS` as the second argument to the routine macro, as follows:

```
_NewPtr ,SYS
```

If you want to clear the bytes of a block of memory to 0 when you allocate it with the `NewPtr` function, set bit 9 of the routine trap word. You can usually do this by supplying the word `CLEAR` as the second argument to the routine macro, as follows:

```
_NewPtr ,CLEAR
```

You can combine `SYS` and `CLEAR` in the same macro call, but `SYS` must come first.

```
_NewPtr ,SYS,CLEAR
```

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory

NewPtrSys

You can use the `NewPtrSys` function to allocate a nonrelocatable block of memory of a specified size in the system heap.

```
FUNCTION NewPtrSys (logicalSize: Size): Ptr;
```

`logicalSize`

The requested size (in bytes) of the nonrelocatable block.

DESCRIPTION

The `NewPtrSys` function works much as the `NewPtr` function does, but attempts to allocate the requested block in the system heap zone instead of in the current heap zone.

RESULT CODES

noErr	0	No error
memFullErr	-108	Not enough memory

NewPtrClear

You can use the `NewPtrClear` function to allocate prezeroed memory in a nonrelocatable block of a specified size.

```
FUNCTION NewPtrClear (logicalSize: Size): Ptr;
```

logicalSize

The requested size (in bytes) of the nonrelocatable block.

DESCRIPTION

The `NewPtrClear` function works much as the `NewPtr` function does, but sets all bytes in the new block to 0 instead of leaving the contents of the block undefined.

Currently, `NewPtrClear` clears the block one byte at a time. For a large block, it might be faster to clear the block manually a long word at a time.

RESULT CODES

noErr	0	No error
memFullErr	-108	Not enough memory

NewPtrSysClear

You can use the `NewPtrSysClear` function to allocate, in the system heap, prezeroed memory in a nonrelocatable block of a specified size.

```
FUNCTION NewPtrSysClear (logicalSize: Size): Ptr;
```

logicalSize

The requested size (in bytes) of the nonrelocatable block.

DESCRIPTION

The `NewPtrSysClear` function works much as the `NewPtr` function does, but attempts to allocate the requested block in the system heap zone instead of in the current heap zone. Also, it sets all bytes in the new block to 0 instead of leaving the contents of the block undefined.

RESULT CODES

noErr	0	No error
memFullErr	-108	Not enough memory

DisposePtr

When you are completely done with a nonrelocatable block, call the `DisposePtr` procedure to free it for other uses.

```
PROCEDURE DisposePtr (p: Ptr);
```

`p` A pointer to the nonrelocatable block you want to dispose of.

DESCRIPTION

The `DisposePtr` procedure releases the memory occupied by the nonrelocatable block specified by `p`.

▲ WARNING

After a call to `DisposePtr`, all pointers to the released block become invalid and should not be used again. Any subsequent use of a pointer to the released block might cause a system error. ▲

SPECIAL CONSIDERATIONS

Because `DisposePtr` purges memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `DisposePtr` are

Registers on entry

A0 Pointer to the nonrelocatable block to be disposed of

Registers on exit

D0 Result code

RESULT CODES

noErr	0	No error
memWZErr	-111	Attempt to operate on a free block

Changing the Sizes of Relocatable and Nonrelocatable Blocks

You can use the `GetHandleSize` function and the `SetHandleSize` procedure to find out and change the logical size of a relocatable block, and you can use the `GetPtrSize`

function and the `SetPtrSize` procedure to find out and change the logical size of a nonrelocatable block.

GetHandleSize

You can use the `GetHandleSize` function to find out the logical size of the relocatable block corresponding to a handle.

```
FUNCTION GetHandleSize (h: Handle): Size;
```

`h` A handle to a relocatable block.

DESCRIPTION

The `GetHandleSize` function returns the logical size, in bytes, of the relocatable block whose handle is `h`. In case of an error, `GetHandleSize` returns 0.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `GetHandleSize` are

Registers on entry

A0 Handle to the relocatable block

Registers on exit

D0 If ≥ 0 , number of bytes in relocatable
 block

 If < 0 , result code

The trap dispatcher sets the condition codes before returning from a trap by testing the low-order word of register `D0` with a `TST.W` instruction. Because the block size returned in `D0` by `_GetHandleSize` is a full 32-bit long word, the word-length test sets the condition codes incorrectly in this case. To branch on the contents of `D0`, use your own `TST.L` instruction on return from the trap to test the full 32 bits of the register.

SPECIAL CONSIDERATIONS

You shouldn't call `GetHandleSize` at interrupt time because the heap might be in an inconsistent state.

RESULT CODES

<code>noErr</code>	0	No error
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memWZErr</code>	-111	Attempt to operate on a free block

SetHandleSize

You can use the `SetHandleSize` procedure to change the logical size of the relocatable block corresponding to a handle.

```
PROCEDURE SetHandleSize (h: Handle; newSize: Size);
```

`h` A handle to a relocatable block.
`newSize` The desired new logical size, in bytes, of the relocatable block.

DESCRIPTION

The `SetHandleSize` procedure attempts to change the logical size of the relocatable block whose handle is `h`. The new logical size is specified by `newSize`. `SetHandleSize` might need to move the relocatable block to obtain enough space for the resized block. Thus, for best results you should unlock a block before resizing it.

An attempt to increase the size of a locked block might fail, because of blocks above and below it that are either nonrelocatable or locked. You should be prepared for this possibility.

SPECIAL CONSIDERATIONS

Because `SetHandleSize` allocates memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `SetHandleSize` are

Registers on entry

A0 Handle to the relocatable block
D0 Desired new size of relocatable block

Registers on exit

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memWZErr</code>	-111	Attempt to operate on a free block

SEE ALSO

Instead of using the `SetHandleSize` procedure to set the size of a handle to 0, you can use the `EmptyHandle` procedure, described on page 6-364.

GetPtrSize

You can use the `GetPtrSize` function to find out the logical size of the nonrelocatable block corresponding to a pointer.

```
FUNCTION GetPtrSize (p: Ptr): Size;
```

`p` A pointer to a nonrelocatable block.

DESCRIPTION

The `GetPtrSize` function returns the logical size, in bytes, of the nonrelocatable block pointed to by `p`. In case of an error, `GetPtrSize` returns 0.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `GetPtrSize` are

Registers on entry

A0 Pointer to the nonrelocatable block

Registers on exit

D0 If ≥ 0 , number of bytes in nonrelocatable block

 If < 0 , result code

The trap dispatcher sets the condition codes before returning from a trap by testing the low-order word of register D0 with a `TST.W` instruction. Because the block size returned in D0 by `_GetPtrSize` is a full 32-bit long word, the word-length test sets the condition codes incorrectly in this case. To branch on the contents of D0, use your own `TST.L` instruction on return from the trap to test the full 32 bits of the register.

RESULT CODES

<code>noErr</code>	0	No error
<code>memWZErr</code>	-111	Attempt to operate on a free block

SetPtrSize

You can use the `SetPtrSize` procedure to change the logical size of the nonrelocatable block corresponding to a pointer.

```
PROCEDURE SetPtrSize (p: Ptr; newSize: Size);
```

<code>p</code>	A pointer to a nonrelocatable block.
<code>newSize</code>	The desired new logical size, in bytes, of the nonrelocatable block.

DESCRIPTION

The `SetPtrSize` procedure attempts to change the logical size of the nonrelocatable block pointed to by `p`. The new logical size is specified by `newSize`.

An attempt to increase the size of a nonrelocatable block might fail because of a block above it that is either nonrelocatable or locked. You should be prepared for this possibility.

SPECIAL CONSIDERATIONS

Because `SetPtrSize` allocates memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `SetPtrSize` are

Registers on entry

A0	Pointer to the nonrelocatable block
D0	Desired new size of nonrelocatable block

Registers on exit

D0	Result code
----	-------------

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory
<code>memWZErr</code>	-111	Attempt to operate on a free block

Setting the Properties of Relocatable Blocks

A relocatable block can be either locked or unlocked and either purgeable or unpurgeable. In addition, it can have its resource bit either set or cleared. To determine the state of any of these properties, use the `HGetState` function. To change these properties, use the `HLock`, `HUnlock`, `HPurge`, `HNoPurge`, `HSetRBit`, and `HClrRBit` procedures. To restore these properties, use the `HSetState` procedure.

▲ WARNING

Be sure to use these procedures to get and set the properties of relocatable blocks. In particular, do not rely on the structure of master pointers, because their structure in 24-bit mode is different from their structure in 32-bit mode. ▲

HGetState

You can use the `HGetState` function to get the current properties of a relocatable block (perhaps so that you can change and then later restore those properties).

```
FUNCTION HGetState (h: Handle): SignedByte;
```

`h` A handle to a relocatable block.

DESCRIPTION

The `HGetState` function returns a signed byte containing the flags of the master pointer for the given handle. You can save this byte, change the state of any of the flags using the routines described on page 6-358 through page 6-363, and then restore their original states by passing the byte to the `HSetState` procedure, described next.

You can use bit-manipulation functions on the returned signed byte to determine the value of a given attribute. Currently the following bits are used:

Bit	Meaning
0-4	Reserved
5	Set if relocatable block is a resource
6	Set if relocatable block is purgeable
7	Set if relocatable block is locked

If an error occurs during an attempt to get the state flags of the specified relocatable block, `HGetState` returns the low-order byte of the result code as its function result. For

example, if the handle `h` points to a master pointer whose value is `NIL`, then the signed byte returned by `HGetState` will contain the value `-109`.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HGetState` are

Registers on entry

A0 Handle whose properties you want to get

Registers on exit

D0 Byte containing
 flags

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>nilHandleErr</code>	<code>-109</code>	<code>NIL</code> master pointer
<code>memWZErr</code>	<code>-111</code>	Attempt to operate on a free block

HSetState

You can use the `HSetState` procedure to restore properties of a block after a call to `HGetState`.

```
PROCEDURE HSetState (h: Handle; flags: SignedByte);
```

`h` A handle to a relocatable block.
`flags` A signed byte specifying the properties to which you want to set the relocatable block.

DESCRIPTION

The `HSetState` procedure restores to the handle `h` the properties specified in the `flags` signed byte. See the description of the `HGetState` function for a list of the currently used bits in that byte. Because additional bits of the `flags` byte could become significant in future versions of system software, use `HSetState` only with a byte returned by `HGetState`. If you need to set two or three properties of a relocatable block at once, it is better to use the procedures that set individual properties than to manipulate the bits returned by `HGetState` and then call `HSetState`.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HSetState` are

Registers on entry

- A0 Handle whose properties you want to set
- D0 Byte containing flags indicating the handle's new properties

Registers on exit

- D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memWZErr</code>	-111	Attempt to operate on a free block

HLock

You can use the `HLock` procedure to lock a relocatable block so that it does not move in the heap. If you plan to dereference a handle and then allocate, move, or purge memory (or call a routine that does so), then you should lock the handle before using the dereferenced handle.

```
PROCEDURE HLock (h: Handle);
```

`h` A handle to a relocatable block.

DESCRIPTION

The `HLock` procedure locks the relocatable block to which `h` is a handle, preventing it from being moved within its heap zone. If the block is already locked, `HLock` does nothing.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HLock` are

Registers on entry

- A0 Handle to lock

Registers on exit

- D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memWZErr</code>	-111	Attempt to operate on a free block

SEE ALSO

If you plan to lock a relocatable block for long periods of time, you can prevent fragmentation by ensuring that the block is as low as possible in the heap zone. To do this, see the description of the `ReserveMem` procedure on page 6-368.

If you plan to lock a relocatable block for short periods of time, you can prevent heap fragmentation by moving the block to the top of the heap zone before locking. For more information, see the description of the `MoveHHI` procedure on page 6-369.

HUnlock

You can use the `HUnlock` procedure to unlock a relocatable block so that it is free to move in its heap zone.

```
PROCEDURE HUnlock (h: Handle);
```

`h` A handle to a relocatable block.

DESCRIPTION

The `HUnlock` procedure unlocks the relocatable block to which `h` is a handle, allowing it to be moved within its heap zone. If the block is already unlocked, `HUnlock` does nothing.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for HUnlock are

Registers on entry

A0 Handle to unlock

Registers on exit

D0 Result code

RESULT CODES

noErr	0	No error
nilHandleErr	-109	NIL master pointer
memWZErr	-111	Attempt to operate on a free block

HPurge

You can use the HPurge procedure to mark a relocatable block so that it can be purged if a memory request cannot be fulfilled after compaction.

```
PROCEDURE HPurge (h: Handle);
```

h A handle to a relocatable block.

DESCRIPTION

The HPurge procedure makes the relocatable block to which h is a handle purgeable. If the block is already purgeable, HPurge does nothing.

The Memory Manager might purge the block when it needs to purge the heap zone containing the block to satisfy a memory request. A direct call to the PurgeMem procedure or the MaxMem function would also purge blocks marked as purgeable.

Once you mark a relocatable block as purgeable, you should make sure that handles to the block are not empty before you access the block. If they are empty, you must reallocate space for the block and recopy the block's data from another source, such as a resource file, before using the information in the block.

If the block to which h is a handle is locked, HPurge does not unlock the block but does mark it as purgeable. If you later call HUnlock on h, the block is subject to purging.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HPurge` are

Registers on entry

A0 Handle to make purgeable

Registers on exit

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memWZErr</code>	-111	Attempt to operate on a free block

SEE ALSO

If the Memory Manager has purged a block, you can reallocate space for it by using the `ReallocateHandle` procedure, described on page 6-365.

You can immediately free the space taken by a handle without disposing of it by calling `EmptyHandle`. This procedure, described on page 6-364, does not require that the block be purgeable.

HNoPurge

You can use the `HNoPurge` procedure to mark a relocatable block so that it cannot be purged.

```
PROCEDURE HNoPurge (h: Handle);
```

`h` A handle to a relocatable block.

DESCRIPTION

The `HNoPurge` procedure makes the relocatable block to which `h` is a handle unpurgeable. If the block is already unpurgeable, `HNoPurge` does nothing.

The `HNoPurge` procedure does not reallocate memory for a handle if it has already been purged.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HNoPurge` are

Registers on entry

A0 Handle to make un purgeable

Registers on exit

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memWZErr</code>	-111	Attempt to operate on a free block

SEE ALSO

If you want to reallocate memory for a relocatable block that has already been purged, you can use the `ReallocateHandle` procedure, described on page 6-365.

HSetRBit

You can use the `HSetRBit` procedure to set the resource flag of a relocatable block. The Resource Manager uses this routine extensively, but you should never need to use it.

```
PROCEDURE HSetRBit (h: Handle);
```

`h` A handle to a relocatable block.

DESCRIPTION

The `HSetRBit` procedure sets the resource flag of the relocatable block to which `h` is a handle. It does nothing if the flag is already set.

▲ WARNING

When the resource flag is set, the Resource Manager identifies the associated relocatable block as belonging to a resource. This can cause problems if that block wasn't actually read from a resource. ▲

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HSetRBit` are

Registers on entry

A0 Handle whose resource flag you want to set

Registers on exit

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memWZErr</code>	-111	Attempt to operate on a free block

HClrRBit

You can use the `HClrRBit` procedure to clear the resource flag of a relocatable block. The Resource Manager uses this routine extensively, but you probably won't need to use it.

```
PROCEDURE HClrRBit (h: Handle);
```

h A handle to a relocatable block.

DESCRIPTION

The `HClrRBit` procedure clears the resource flag of a relocatable block. It does nothing if the flag is already cleared.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HClrRBit` are

Registers on entry

A0 Handle whose resource flag you want to clear

Registers on exit

D0 Result code

Memory Manager

RESULT CODES

<code>noErr</code>	0	No error
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memWZErr</code>	-111	Attempt to operate on a free block

SEE ALSO

To disassociate the data in a resource handle from the resource file, you should use the Resource Manager procedure `DetachResource` instead of this procedure.

Managing Relocatable Blocks

The Memory Manager provides routines that allow you to purge and later reallocate space for relocatable blocks, recreate handles to relocatable blocks if you have access to their master pointers, and control where in their heap zone relocatable blocks are located.

To free the memory taken up by a relocatable block without releasing the master pointer to the block for other uses, use the `EmptyHandle` procedure. To reallocate space for a handle that you have emptied or the Memory Manager has purged, use the `ReallocateHandle` procedure.

If, because you have dereferenced a handle, you no longer have access to it but do have access to its master pointer, you can use the `RecoverHandle` function to recreate the handle.

To ensure that a relocatable block that you plan to lock for short or long periods of time does not cause heap fragmentation, use the `MoveHHi` and the `ReserveMem` procedures, respectively.

EmptyHandle

The `EmptyHandle` procedure allows you to free memory taken by a relocatable block without freeing the relocatable block's master pointer for other uses.

```
PROCEDURE EmptyHandle (h: Handle);
```

`h` A handle to a relocatable block.

DESCRIPTION

The `EmptyHandle` procedure purges the relocatable block whose handle is `h` and sets the handle's master pointer to NIL. The block whose handle is `h` must be unlocked but need not be purgeable.

Note

If there are multiple handles to the relocatable block, then calling the `EmptyHandle` procedure empties them all, because all of the handles share a common master pointer. When you later use `ReallocateHandle` to reallocate space for the block, the master pointer is updated, and all of the handles reference the new block correctly. ♦

SPECIAL CONSIDERATIONS

Because `EmptyHandle` purges memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `EmptyHandle` are

Registers on entry

A0 Handle to relocatable block

Registers on exit

A0 Handle to relocatable block

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>memWZErr</code>	-111	Attempt to operate on a free block
<code>memPurErr</code>	-112	Attempt to purge a locked block

SEE ALSO

To purge all of the blocks in a heap zone that are marked purgeable, use the `PurgeMem` procedure, described on page 6-386.

To free the memory taken up by a relocatable block and release the block's master pointer for other uses, use the `DisposeHandle` procedure, described on page 6-347.

ReallocateHandle

To recover space for a relocatable block that you have emptied or the Memory Manager has purged, use the `ReallocateHandle` procedure.

PROCEDURE `ReallocateHandle` (`h`: `Handle`; `logicalSize`: `Size`);

`h` A handle to a relocatable block.

`logicalSize` The desired new logical size (in bytes) of the relocatable block.

DESCRIPTION

The `ReallocateHandle` procedure allocates a new relocatable block with a logical size of `logicalSize` bytes. It updates the handle `h` by setting its master pointer to point to the new block. The new block is unlocked and unpurgeable.

Usually you use `ReallocateHandle` to reallocate space for a block that you have emptied or the Memory Manager has purged. If the handle references an existing block, `ReallocateHandle` releases that block before creating a new one.

Note

To reallocate space for a resource that has been purged, you should call `LoadResource`, not `ReallocateHandle`. ♦

If many handles reference a single purged, relocatable block, you need to call `ReallocateHandle` on just one of them.

In case of an error, `ReallocateHandle` neither allocates a new block nor changes the master pointer to which handle `h` points.

SPECIAL CONSIDERATIONS

Because `ReallocateHandle` might purge and allocate memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `ReallocateHandle` are

Registers on entry

A0 Handle for new relocatable block
D0 Desired logical size, in bytes, of new block

Registers on exit

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>memROZErr</code>	-99	Heap zone is read-only
<code>memFullErr</code>	-108	Not enough memory
<code>memWZErr</code>	-111	Attempt to operate on a free block
<code>memPurErr</code>	-112	Attempt to purge a locked block

SEE ALSO

Because `ReallocateHandle` releases any existing relocatable block referenced by the handle `h` before allocating a new one, it does not provide an efficient technique for resizing relocatable blocks. To do that, use the `SetHandleSize` procedure, described on page 6-353.

RecoverHandle

The Memory Manager does not allow you to change relocatable blocks into nonrelocatable blocks, or vice-versa. However, if you no longer have access to a handle but still have access to its master pointer, you can use the `RecoverHandle` function to recreate a handle to the relocatable block referenced by the master pointer.

```
FUNCTION RecoverHandle (p: Ptr): Handle;
```

`p` The master pointer to a relocatable block.

DESCRIPTION

The `RecoverHandle` function returns a handle to the relocatable block pointed to by `p`. If `p` doesn't point to a valid block, the results of `RecoverHandle` are undefined.

SPECIAL CONSIDERATIONS

Even though `RecoverHandle` does not move or purge memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `RecoverHandle` are

Registers on entry

A0 Master pointer

Registers on exit

A0 Handle to master pointer's relocatable block

D0 Unchanged

Unlike most other Memory Manager routines, `RecoverHandle` does not return a result code in register D0; the previous contents of D0 are preserved unchanged.

The result code is, however, returned by `MemError`.

The `RecoverHandle` function looks only in the current heap zone for the relocatable block pointed to by the parameter `p`. If you want to use the `RecoverHandle` function to recover a handle for a relocatable block in the system heap, set bit 10 of the routine trap word. In most development systems, you can do this by supplying the word `SYS` as the second argument to the routine macro, as follows:

```
_RecoverHandle ,SYS
```

RESULT CODES

<code>noErr</code>	0	No error
<code>memBCErr</code>	-115	Block check failed

ReserveMem

Use the `ReserveMem` procedure when you allocate a relocatable block that you intend to lock for long periods of time. This helps prevent heap fragmentation because it reserves space for the block as close to the bottom of the heap as possible. Consistent use of `ReserveMem` for this purpose ensures that all locked, relocatable blocks and nonrelocatable blocks are together at the bottom of the heap zone and thus do not prevent unlocked relocatable blocks from moving about the zone.

```
PROCEDURE ReserveMem (cbNeeded: Size);
```

`cbNeeded` The number of bytes to reserve near the bottom of the heap.

DESCRIPTION

The `ReserveMem` procedure attempts to create free space for a block of `cbNeeded` contiguous logical bytes at the lowest possible position in the current heap zone. It pursues every available means of placing the block as close as possible to the bottom of the zone, including moving other relocatable blocks upward, expanding the zone (if possible), and purging blocks from it.

Because `ReserveMem` does not actually allocate the block, you must combine calls to `ReserveMem` with calls to the `NewHandle` function.

Do not use the `ReserveMem` procedure for a relocatable block you intend to lock for only a short period of time. If you do so and then allocate a nonrelocatable block above it, the relocatable block becomes trapped under the nonrelocatable block when you unlock that relocatable block.

Note

It isn't necessary to call `ReserveMem` to reserve space for a nonrelocatable block, because the `NewPtr` function calls it automatically. Also, you do not need to call `ReserveMem` to reserve memory before you load a locked resource into memory, because the Resource Manager calls `ReserveMem` automatically. ♦

SPECIAL CONSIDERATIONS

Because the `ReserveMem` procedure could move and purge memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `ReserveMem` are

Registers on entry

D0 Number of bytes to reserve

Registers on exit

D0 Result code

The `ReserveMem` procedure reserves memory in the current heap zone. If you want to reserve memory in the system heap zone rather than in the current heap zone, set bit 10 of the routine trap word. In most development systems, you can do this by supplying the word `SYS` as the second argument to the routine macro, as follows:

```
_ResrvMem ,SYS
```

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory

ReserveMemSys

If you plan to lock a relocatable block for long periods of time in the system heap zone, use the `ReserveMemSys` procedure to reserve space for the block as low in the system heap as possible.

```
PROCEDURE ReserveMemSys (cbNeeded: Size);
```

`cbNeeded` The number of bytes to reserve near the bottom of the system heap.

DESCRIPTION

The `ReserveMemSys` procedure works much as the `ReserveMem` procedure does, but reserves memory in the system heap zone rather than in the current heap zone.

MoveHHi

If you plan to lock a relocatable block for a short period of time, use the `MoveHHi` procedure, which moves the block to the top of the heap and thus helps prevent heap fragmentation.

```
PROCEDURE MoveHHi (h: Handle);
```

Memory Manager

`h` A handle to a relocatable block.

DESCRIPTION

The `MoveHHi` procedure attempts to move the relocatable block referenced by the handle `h` upward until it reaches a nonrelocatable block, a locked relocatable block, or the top of the heap.

▲ WARNING

If you call `MoveHHi` to move a handle to a resource that has its `resChanged` bit set, the Resource Manager updates the resource by using the `WriteResource` procedure to write the contents of the block to disk. If you want to avoid this behavior, call the Resource Manager procedure `SetResPurge(FALSE)` before you call `MoveHHi`, and then call `SetResPurge(TRUE)` to restore the default setting. ▲

By using the `MoveHHi` procedure on relocatable blocks you plan to allocate for short periods of time, you help prevent islands of immovable memory from accumulating in (and thus fragmenting) the heap.

Do not use the `MoveHHi` procedure to move blocks you plan to lock for long periods of time. The `MoveHHi` procedure moves such blocks to the top of the heap, perhaps preventing other blocks already at the top of the heap from moving down once they are unlocked. Instead, use the `ReserveMem` procedure before allocating such blocks, thus keeping them in the bottom partition of the heap, where they do not prevent relocatable blocks from moving.

If you frequently lock a block for short periods of time and find that calling `MoveHHi` each time slows down your application, you might consider leaving the block always locked and calling the `ReserveMem` procedure before allocating it.

Once you move a block to the top of the heap, be sure to lock it if you do not want the Memory Manager to move it back to the middle partition as soon as it can. (The `MoveHHi` procedure cannot move locked blocks; be sure to lock blocks after, not before, calling `MoveHHi`.)

Note

Using the `MoveHHi` procedure without taking other precautionary measures to prevent heap fragmentation is useless, because even one small nonrelocatable or locked relocatable block in the middle of the heap might prevent `MoveHHi` from moving blocks to the top of the heap. ♦

SPECIAL CONSIDERATIONS

Because the `MoveHHi` procedure moves memory, you should not call it at interrupt time.

Don't call `MoveHHi` on blocks in the system heap. Don't call `MoveHHi` from a desk accessory.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `MoveHHi` are

Registers on entry

A0 Handle to move

Registers on exit

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memLockedErr</code>	-117	Block is locked

HLockHi

You can use the `HLockHi` procedure to move a relocatable block to the top of the heap and lock it.

```
PROCEDURE HLockHi (h: Handle);
```

`h` A handle to a relocatable block.

DESCRIPTION

The `HLockHi` procedure attempts to move the relocatable block referenced by the handle `h` upward until it reaches a nonrelocatable block, a locked relocatable block, or the top of the heap. Then `HLockHi` locks the block.

The `HLockHi` procedure is simply a convenient replacement for the pair of procedures `MoveHHi` and `HLock`.

SPECIAL CONSIDERATIONS

Because the `HLockHi` procedure moves memory, you should not call it at interrupt time.

Don't call `HLockHi` on blocks in the system heap. Don't call `HLockHi` from a desk accessory.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HLockHi` are

Registers on entry

A0 Handle to move and lock

Registers on exit

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memWZErr</code>	-111	Attempt to operate on a free block
<code>memLockedErr</code>	-117	Block is locked

Manipulating Blocks of Memory

The Memory Manager provides three routines for copying blocks of memory referenced by pointers. To copy a block of memory to a nonrelocatable block, use the `BlockMove` procedure. To copy to a new relocatable block, use the `PtrToHand` function. To copy to an existing relocatable block, use the `PtrToXHand` function. If you want to use any of these routines to copy memory you access with a handle, you must first dereference and lock the handle. A fourth routine, `HandToHand`, allows you to copy information from one handle to another.

To concatenate blocks of memory, you can use the `HandAndHand` and `PtrAndHand` functions.

BlockMove

To copy a sequence of bytes from one location in memory to another, you can use the `BlockMove` procedure.

```
PROCEDURE BlockMove (sourcePtr, destPtr: Ptr; byteCount: Size);
```

`sourcePtr` The address of the first byte to copy.

`destPtr` The address of the first byte to copy to.

`byteCount` The number of bytes to copy. If the value of `byteCount` is 0, `BlockMove` does nothing.

DESCRIPTION

The `BlockMove` procedure moves a block of `byteCount` consecutive bytes from the address designated by `sourcePtr` to that designated by `destPtr`. It updates no pointers.

The `BlockMove` procedure works correctly even if the source and destination blocks overlap.

SPECIAL CONSIDERATIONS

You can safely call `BlockMove` at interrupt time. Even though it moves memory, `BlockMove` does not move relocatable blocks, but simply copies bytes.

The `BlockMove` procedure currently flushes the processor caches whenever the number of bytes to be moved is greater than 12. This behavior can adversely affect your application's performance. You might want to avoid calling `BlockMove` to move small amounts of data in memory if there is no possibility of moving stale data or instructions. For more information about stale data and instructions, see the discussion of the processor caches in the chapter "Memory Management Utilities" in this book.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `BlockMove` are

Registers on entry

- A0 Pointer to source
- A1 Pointer to destination
- D0 Number of bytes to copy

Registers on exit

- D0 Result code

RESULT CODE

- `noErr` 0 No error

PtrToHand

To copy data referenced by a pointer to a new relocatable block, use the `PtrToHand` function.

```
FUNCTION PtrToHand (srcPtr: Ptr; VAR dstHndl: Handle;
                    size: LongInt): OSErr;
```

- `srcPtr` The address of the first byte to copy.
- `dstHndl` A handle for which you have not yet allocated any memory. The `PtrToHand` function allocates memory for the handle and copies `size` bytes beginning at `srcPtr` into it.
- `size` The number of bytes to copy.

DESCRIPTION

The `PtrToHand` function returns, in `dstHndl`, a newly created handle to a copy of the number of bytes specified by the `size` parameter, beginning at the location specified by `srcPtr`. The `dstHndl` parameter must be a handle variable that is not empty and is not a handle to an allocated block of size 0.

SPECIAL CONSIDERATIONS

Because `PtrToHand` allocates memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `PtrToHand` are

Registers on entry

A0 Pointer to source
D0 Number of bytes to copy

Registers on exit

A0 Destination handle
D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory

SEE ALSO

You can use the `PtrToHand` function to copy data from one handle to a new handle if you dereference and lock the source handle. However, if you want to copy all of the data from one handle to another, the `HandToHand` function (described on page 6-375) is more efficient.

PtrToXHand

To copy data referenced by a pointer to an already existing relocatable block, use the `PtrToXHand` function.

```
FUNCTION PtrToXHand (srcPtr: Ptr; dstHndl: Handle; size: LongInt):
    OSErr;
```

`srcPtr` The address of the first byte to copy.

Memory Manager

<code>dstHndl</code>	A handle to an already existing relocatable block to which to copy <code>size</code> bytes, beginning at <code>srcPtr</code> .
<code>size</code>	The number of bytes to copy.

DESCRIPTION

The `PtrToXHand` function makes the existing handle, specified by `dstHndl`, a handle to a copy of the number of bytes specified by the `size` parameter, beginning at the location specified by `srcPtr`.

SPECIAL CONSIDERATIONS

Because `PtrToXHand` affects memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `PtrToXHand` are

Registers on entry

<code>A0</code>	Pointer to source
<code>A1</code>	Handle to destination
<code>D0</code>	Number of bytes to copy

Registers on exit

<code>A0</code>	Handle to destination
<code>D0</code>	Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memWZErr</code>	-111	Attempt to operate on a free block

HandToHand

Use the `HandToHand` function to copy all of the data from one relocatable block to a new relocatable block.

```
FUNCTION HandToHand (VAR theHndl: Handle): OSErr;
```

<code>theHndl</code>	On entry, a handle to the relocatable block whose data is to be copied. On exit, a handle to a new relocatable block whose data duplicates that of the original.
----------------------	--

Memory Manager

DESCRIPTION

The `HandToHand` function attempts to copy the information in the relocatable block to which `theHndl` is a handle; if successful, `HandToHand` returns a handle to the new relocatable block in `theHndl`. The new relocatable block is created in the same heap zone as the original block (which might not be the current heap zone).

Because `HandToHand` replaces its input parameter with the new handle, you should retain the original value of the input parameter somewhere else, or you won't be able to access it. Here is an example:

```
VAR
    original, copy: Handle;
    myErr: OSerr;
...
    copy := original;           {both handles access same block}
    myErr := HandToHand(copy); {copy now points to copy of block}
```

SPECIAL CONSIDERATIONS

If successful in creating a new relocatable block, the `HandToHand` function does not duplicate the properties of the original block. The new block is unlocked, unpurgeable, and not a resource. You might need to call `HLock`, `HPurge`, or `HSetRBit` (or the combination of `HGetState` and `HSetState`) to adjust the properties of the new block.

Because `HandToHand` allocates memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HandToHand` are

Registers on entry

A0 Handle to original data

Registers on exit

A0 Handle to copy of data

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memWZErr</code>	-111	Attempt to operate on a free block

SEE ALSO

If you want to copy only part of a relocatable block into a new relocatable block, use the `PtrToHand` function, described on page 6-373, after locking and dereferencing a handle to the relocatable block to be copied.

HandAndHand

Use the `HandAndHand` function to concatenate two relocatable blocks.

```
FUNCTION HandAndHand (aHndl, bHndl: Handle): OSErr;
```

<code>aHndl</code>	A handle to the first relocatable block, whose contents do not change but are concatenated to the end of the second relocatable block.
<code>bHndl</code>	A handle to the second relocatable block, whose size the Memory Manager expands so that it can concatenate the information from <code>aHndl</code> to the end of the contents of this block.

DESCRIPTION

The `HandAndHand` function concatenates the information from the relocatable block to which `aHndl` is a handle onto the end of the relocatable block to which `bHndl` is a handle. The `aHndl` variable remains unchanged.

▲ **WARNING**

The `HandAndHand` function dereferences the handle `aHndl`. You must call the `HLock` procedure to lock the block before calling `HandAndHand`. Afterward, you can call the `HUnlock` procedure to unlock it. Alternatively, you can save the block's original state by calling the `HGetState` function, lock the block by calling `HLock`, and then restore the original settings by calling `HSetState`. ▲

SPECIAL CONSIDERATIONS

Because `HandAndHand` moves memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HandAndHand` are

Registers on entry

A0	Handle to be concatenated
A1	Handle to contain itself, data from A0's handle

Registers on exit

A0	Handle to concatenated data
D0	Result code

Memory Manager

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memWZErr</code>	-111	Attempt to operate on a free block

PtrAndHand

Use the `PtrAndHand` function to concatenate part or all of a memory block to the end of a relocatable block.

```
FUNCTION PtrAndHand (pntr: Ptr; hndl: Handle; size: LongInt):
                    OSErr;
```

<code>pntr</code>	A pointer to the beginning of the data that the Memory Manager is to concatenate onto the end of the relocatable block.
<code>hndl</code>	A handle to the relocatable block, whose size the Memory Manager expands so that it can concatenate the information from <code>pntr</code> onto the end of this block.
<code>size</code>	The number of bytes of the block referenced by <code>pntr</code> to be copied.

DESCRIPTION

The `PtrAndHand` function takes the number of bytes specified by the `size` parameter, beginning at the location specified by `pntr`, and concatenates them onto the end of the relocatable block to which `hndl` is a handle.

The contents of the source block remain unchanged.

SPECIAL CONSIDERATIONS

Because `PtrAndHand` allocates memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `PtrAndHand` are

Registers on entry

A0	Pointer to data to copy
A1	Handle to relocatable block at whose end the copied data concatenated
A2	Number of bytes to concatenate

Registers on exit

A0	Handle to now-concatenated relocatable block
D0	Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memWZErr</code>	-111	Attempt to operate on a free block

Assessing Memory Conditions

The Memory Manager provides four routines to test how much memory is available, one routine used after memory operations to determine if an error occurred, and one routine to determine the location in memory of the top of your application's partition.

To determine the total amount of free space in the current heap zone or the size of the maximum block that could be obtained after compacting the heap, use the `FreeMem` and `MaxBlock` functions, respectively. To determine what those values would be after a purge of the heap zone, call the `PurgeSpace` procedure. Finally, to find out how much your stack can grow before it collides with the heap, use the `StackSpace` function.

To find out whether a Memory Manager operation finished successfully, use the `MemError` function.

FreeMem

By calling the `FreeMem` function, you can find out the total amount of free space, in bytes, in the current heap zone.

```
FUNCTION FreeMem: LongInt;
```

DESCRIPTION

The `FreeMem` function returns the total amount of free space (in bytes) in the current heap zone. Note that it usually isn't possible to allocate a block of that size, because of heap fragmentation due to nonrelocatable or locked blocks.

SPECIAL CONSIDERATIONS

Even though `FreeMem` does not move or purge memory, you should not call it at interrupt time because the heap might be in an inconsistent state.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `FreeMem` are

Registers on exit

D0 Number of bytes available in heap zone

Memory Manager

The `FreeMem` function reports the number of free bytes in the current heap zone. If you want to know how many bytes are available in the system heap zone rather than in the current heap zone, set bit 10 of the routine trap word. In most development systems, you can do this by supplying the word `SYS` as the second argument to the routine macro, as follows:

```
_FreeMem ,SYS
```

RESULT CODES

`noErr` 0 No error

FreeMemSys

To determine how much free space remains in the system heap zone, use the `FreeMemSys` function.

```
FUNCTION FreeMemSys: LongInt;
```

DESCRIPTION

The `FreeMemSys` function works much as the `FreeMem` function does, but returns the total amount of free memory in the system heap zone instead of in the current heap zone.

RESULT CODES

`noErr` 0 No error

MaxBlock

Use the `MaxBlock` function to determine the size of the largest block you could allocate in the current heap zone after a compaction.

```
FUNCTION MaxBlock: LongInt;
```

DESCRIPTION

The `MaxBlock` function returns the maximum contiguous space, in bytes, that you could obtain after compacting the current heap zone. `MaxBlock` does not actually do the compaction.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `MaxBlock` are

Registers on exit

`D0` Size of largest allocatable block

If you want to know the size of the largest allocatable block in the system heap zone, rather than in the current heap zone, set bit 10 of the routine trap word. In most development systems, you can do this by supplying the word `SYS` as the second argument to the routine macro, as follows:

```
_MaxBlock ,SYS
```

RESULT CODES

`noErr` 0 No error

MaxBlockSys

Use the `MaxBlockSys` function to determine the size of the largest block you could allocate in the system heap after a compaction.

```
FUNCTION MaxBlockSys: LongInt;
```

DESCRIPTION

The `MaxBlockSys` function works much as the `MaxBlock` function does, but returns the maximum contiguous space, in bytes, that you could obtain after compacting the system heap. `MaxBlockSys` does not actually do the compaction.

RESULT CODES

`noErr` 0 No error

PurgeSpace

Use the `PurgeSpace` procedure to determine the total amount of free memory and the size of the largest allocatable block after a purge of the heap.

```
PROCEDURE PurgeSpace (VAR total: LongInt; VAR contig: LongInt);
```

`total` On exit, the total amount of free memory in the current heap zone if it were purged.

Memory Manager

`contig` On exit, the size of the largest contiguous block of free memory in the current heap zone if it were purged.

DESCRIPTION

The `PurgeSpace` procedure returns, in the `total` parameter, the total amount of space (in bytes) that could be obtained after a general purge of the current heap zone; this amount includes space that is already free. In the `contig` parameter, `PurgeSpace` returns the size of the largest allocatable block in the current heap zone that could be obtained after a purge of the zone.

The `PurgeSpace` procedure does not actually purge the current heap zone.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `PurgeSpace` are

Registers on exit

A0 Maximum number of contiguous bytes after purge

D0 Total free memory after purge

If you want to test the system heap zone instead of the current zone, set bit 10 of the routine trap word. In most development systems, you can do this by supplying the word `SYS` as the second argument to the routine macro, as follows:

```
_PurgeSpace ,SYS
```

RESULT CODES

`noErr` 0 No error

StackSpace

Use the `StackSpace` function to find out how much space there is between the bottom of the stack and the top of the application heap.

```
FUNCTION StackSpace: LongInt;
```

DESCRIPTION

The `StackSpace` function returns the current amount of stack space (in bytes) between the current stack pointer and the application heap at the instant of return from the trap.

SPECIAL CONSIDERATIONS

Ordinarily, you determine the maximum amount of stack space you need before you ship your application. In general, therefore, this routine is useful only during debugging to determine how big to make the stack. However, if your application calls a recursive function that conceivably could call itself many times, that function should keep track of the stack space and take appropriate action if it becomes too low.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `StackSpace` are

Registers on exit

D0 Number of bytes between stack and heap

RESULT CODES

`noErr` 0 No error

MemError

To find out whether your application's last direct call to a Memory Manager routine executed successfully, use the `MemError` function.

```
FUNCTION MemError: OSErr;
```

DESCRIPTION

The `MemError` function returns the result code produced by the last Memory Manager routine your application called directly.

This function is useful during application debugging. You might also use the function as one part of a memory-management scheme to identify instances in which the Memory Manager rejects overly large memory requests by returning the error code `memFullErr`.

▲ WARNING

Do not rely on the `MemError` function as the only component of a memory-management scheme. For example, suppose you call `NewHandle` or `NewPtr` and receive the result code `noErr`, indicating that the Memory Manager was able to allocate sufficient memory. In this case, you have no guarantee that the allocation did not deplete your application's memory reserves to levels so low that simple operations might cause your application to crash. Instead of relying on `MemError`, check before making a memory request that there is enough memory both to fulfill the request and to support essential operations. ▲

ASSEMBLY-LANGUAGE INFORMATION

Because most Memory Manager routines return a result code in register D0, you do not ordinarily need to call the `MemError` function if you program in assembly language. See the description of an individual routine to find out whether it returns a result code in register D0. If not, you can examine the global variable `MemErr`. When `MemError` returns, register D0 contains the result code.

Registers on exit

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Error in parameter list
<code>memROZErr</code>	-99	Operation on a read-only zone
<code>memFullErr</code>	-108	Not enough memory
<code>nilHandleErr</code>	-109	NIL master pointer
<code>memWZErr</code>	-111	Attempt to operate on a free block
<code>memPurErr</code>	-112	Attempt to purge a locked block
<code>memBCErr</code>	-115	Block check failed
<code>memLockedErr</code>	-117	Block is locked

Freeing Memory

The Memory Manager compacts and purges the heap whenever necessary to satisfy requests for memory. You can also compact or purge the heap manually. To compact the current heap zone manually, use the `CompactMem` function. To purge it manually, use the `PurgeMem` procedure. To do both at once, use the `MaxMem` function. To perform the same operations on the system heap zone, use the `CompactMemSys` function, the `PurgeMemSys` procedure, and the `MaxMemSys` function.

Note

Most applications don't need to call the routines described in this section. Normally you should let the Memory Manager compact or purge your application heap. ♦

CompactMem

The Memory Manager compacts the heap for you when you make a memory request that it can't fill. However, you can use the `CompactMem` function to compact the current heap zone manually.

```
FUNCTION CompactMem (cbNeeded: Size): Size;
```

`cbNeeded` The size, in bytes, of the block for which `CompactMem` should attempt to make room.

DESCRIPTION

The `CompactMem` function compacts the current heap zone by moving unlocked, relocatable blocks down until they encounter nonrelocatable blocks or locked, relocatable blocks, but not by purging blocks. It continues compacting until it either finds a contiguous block of at least `cbNeeded` free bytes or has compacted the entire zone.

The `CompactMem` function returns the size, in bytes, of the largest contiguous free block for which it could make room, but it does not actually allocate that block.

To compact the entire heap zone, call `CompactMem(maxSize)`. The Memory Manager defines the constant `maxSize` for the largest contiguous block possible in the 24-bit Memory Manager:

```
CONST
    maxSize                = $800000;           {maximum size of a block}
```

SPECIAL CONSIDERATIONS

Because `CompactMem` moves memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `CompactMem` are

Registers on entry

D0 Size of block to make room for

Registers on exit

D0 Size of largest allocatable block

The `CompactMem` function compacts the current heap zone. If you want to compact the system heap zone rather than the current heap zone, set bit 10 of the routine trap word. In most development systems, you can do this by supplying the word `SYS` as the second argument to the routine macro, as follows:

```
_CompactMem ,SYS
```

RESULT CODES

noErr 0 No error

CompactMemSys

You can use the `CompactMemSys` function to compact the system heap zone manually.

```
FUNCTION CompactMemSys (cbNeeded: Size): Size;
```

Memory Manager

cbNeeded The size in bytes of the block for which `CompactMemSys` should attempt to make room.

DESCRIPTION

The `CompactMemSys` function works much as the `CompactMem` function does, but compacts the system heap instead of the current heap.

RESULT CODES

`noErr` 0 No error

PurgeMem

The Memory Manager purges the heap for you when you make a memory request that it can't fill. However, you can use the `PurgeMem` procedure to purge the current heap zone manually.

```
PROCEDURE PurgeMem (cbNeeded: Size);
```

cbNeeded The size, in bytes, of the block for which `PurgeMem` should attempt to make room.

DESCRIPTION

The `PurgeMem` procedure sequentially purges blocks from the current heap zone until it either allocates a contiguous block of at least `cbNeeded` free bytes or has purged the entire zone. If it purges the entire zone without creating a contiguous block of at least `cbNeeded` free bytes, `PurgeMem` generates a `memFullErr`.

The `PurgeMem` procedure purges only relocatable, unlocked, purgeable blocks.

The `PurgeMem` procedure does not actually attempt to allocate a block of `cbNeeded` bytes.

To purge the entire heap zone, call `PurgeMem(maxSize)`.

SPECIAL CONSIDERATIONS

Because `PurgeMem` purges memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `PurgeMem` are

Registers on entry

D0 Size of block to make room for

Registers on exit

D0 Result code

The `PurgeMem` procedure purges the current heap zone. If you want to purge the system heap zone rather than the current heap zone, set bit 10 of the routine trap word. In most development systems, you can do this by supplying the word `SYS` as the second argument to the routine macro, as follows:

```
_PurgeMem ,SYS
```

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory

PurgeMemSys

You can use the `PurgeMemSys` procedure to purge the system heap manually.

```
PROCEDURE PurgeMemSys (cbNeeded: Size);
```

`cbNeeded` The size, in bytes, of the block for which `PurgeMemSys` should attempt to make room.

DESCRIPTION

The `PurgeMemSys` procedure works much as the `PurgeMem` procedure does, but purges the system heap instead of the current heap.

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory

MaxMem

Use the `MaxMem` function to compact and purge the current heap zone.

```
FUNCTION MaxMem (VAR grow: Size): Size;
```

Memory Manager

grow On exit, the maximum number of bytes by which the current heap zone can grow. After a call to `MaxApplZone`, `MaxMem` always returns 0 in this parameter.

DESCRIPTION

The `MaxMem` function compacts the current heap zone and purges all relocatable, unlocked, and purgeable blocks from the zone. It returns the size, in bytes, of the largest contiguous free block in the zone after the compacting and purging. If the current zone is the original application zone, the `grow` parameter is set to the maximum number of bytes by which the zone can grow. For any other heap zone, `grow` is set to 0. `MaxMem` doesn't actually expand the zone or call the zone's grow-zone function.

SPECIAL CONSIDERATIONS

Because `MaxMem` moves and purges memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `MaxMem` are

Registers on exit

A0 Number of bytes zone can grow
D0 Size in bytes of largest allocatable block

The `MaxMem` function compacts the current heap zone. If you want to compact and purge the system heap zone rather than the current heap zone, set bit 10 of the routine trap word. In most development systems, you can do this by supplying the word `SYS` as the second argument to the routine macro, as follows:

```
_MaxMem ,SYS
```

RESULT CODES

`noErr` 0 No error

MaxMemSys

You can use the `MaxMemSys` function to purge and compact the system heap zone manually.

```
FUNCTION MaxMemSys (VAR grow: Size): Size;
```

grow On exit, the `MaxMemSys` function sets this parameter to 0. Ignore this parameter.

DESCRIPTION

The `MaxMemSys` function works much as the `MaxMem` function does, but compacts and purges the system heap instead of the current heap. It returns the size, in bytes, of the largest block you can allocate in the system heap.

RESULT CODES

`noErr` 0 No error

Grow-Zone Operations

You can implement a grow-zone function that the Memory Manager calls when it cannot fulfill a memory request. You should use the grow-zone function only as a last resort to free memory when all else fails. For explanations of how grow-zone functions work and an example of a memory-management scheme that uses a grow-zone function, see the discussion of low-memory conditions in the chapter “Introduction to Memory Management” in this book.

The `SetGrowZone` procedure specifies which function the Memory Manager should use for the current zone. The grow-zone function should call the `GZSaveHnd` function to receive a handle to a relocatable block that the grow-zone function must not move or purge.

SetGrowZone

To specify a grow-zone function for the current heap zone, pass a pointer to that function to the `SetGrowZone` procedure. Ordinarily, you call this procedure early in the execution of your application.

If you initialize your own heap zones besides the application and system zones, you can alternatively specify a grow-zone function as a parameter to the `InitZone` procedure.

```
PROCEDURE SetGrowZone (growZone: ProcPtr);
```

`growZone` A pointer to the grow-zone function.

DESCRIPTION

The `SetGrowZone` procedure sets the current heap zone’s grow-zone function as designated by the `growZone` parameter. A `NIL` parameter value removes any grow-zone function the zone might previously have had.

The Memory Manager calls the grow-zone function only after exhausting all other avenues of satisfying a memory request, including compacting the zone, increasing its size (if it is the original application zone and is not yet at its maximum size), and purging blocks from it.

Memory Manager

See “Grow-Zone Functions” on page 6-402 for a complete description of a grow-zone function.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `SetGrowZone` are

Registers on entry

A0 Pointer to new grow-zone function

Registers on exit

D0 Result code

RESULT CODES

`noErr` 0 No error

GZSaveHnd

Your grow-zone function must call the `GZSaveHnd` function to obtain a handle to a protected relocatable block that the grow-zone function must not move, purge, or delete.

```
FUNCTION GZSaveHnd: Handle;
```

DESCRIPTION

The `GZSaveHnd` function returns a handle to a relocatable block that the grow-zone function must not move, purge, or delete. It returns `NIL` if there is no such block. The returned handle is a handle to the block of memory being manipulated by the Memory Manager at the time that the grow-zone function is called.

ASSEMBLY-LANGUAGE INFORMATION

You can find the same handle in the global variable `GZRootHnd`.

Allocating Temporary Memory

In system software version 7.0 and later, you can manipulate temporary memory with three routines that are counterparts to other Memory Manager routines. The `TempNewHandle` function allocates a new block of relocatable memory, the `TempFreeMem` function returns the total amount of free memory available for temporary

allocation, and the `TempMaxMem` function compacts the heap zone and returns the size of the largest contiguous block available for temporary allocation.

▲ **WARNING**

You should not call any of these memory-allocation routines at interrupt time. ▲

TempNewHandle

To allocate a new relocatable block of temporary memory, call the `TempNewHandle` function after making sure that there is enough free space to satisfy the request.

```
FUNCTION TempNewHandle (logicalSize: Size;
                       VAR resultCode: OSErr): Handle;
```

`logicalSize`

The requested logical size, in bytes, of the new temporary block of memory.

`resultCode`

On exit, the result code from the function call.

DESCRIPTION

The `TempNewHandle` function returns a handle to a block of size `logicalSize`. If it cannot allocate a block of that size, the function returns `NIL`. Before you use the returned handle, make sure its value is not `NIL`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `TempNewHandle` are

Trap macro	Selector
<code>_OSDispatch</code>	<code>\$001D</code>

SPECIAL CONSIDERATIONS

Because `TempNewHandle` might allocate memory, you should not call it at interrupt time.

Note that `TempNewHandle` returns its result code in a parameter, not through `MemError`.

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory

TempFreeMem

To find out the total amount of memory available for temporary allocation, use the `TempFreeMem` function.

```
FUNCTION TempFreeMem: LongInt;
```

DESCRIPTION

The `TempFreeMem` function returns the total amount of free temporary memory that you could allocate by calling `TempNewHandle`. The returned value is the total number of free bytes. Because these bytes might be dispersed throughout memory, it is ordinarily not possible to allocate a single relocatable block of that size.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `TempFreeMem` are

Trap macro	Selector
<code>_OSDispatch</code>	<code>\$0018</code>

SPECIAL CONSIDERATIONS

Even though `TempFreeMem` does not move or purge memory, you should not call it at interrupt time.

TempMaxMem

To find the size of the largest contiguous block available for temporary allocation, use the `TempMaxMem` function.

```
FUNCTION TempMaxMem (VAR grow: Size): Size;
```

<code>grow</code>	On exit, this parameter always contains 0 after the function call because temporary memory does not come from the application's heap zone, and only that zone can grow. Ignore this parameter.
-------------------	--

DESCRIPTION

The `TempMaxMem` function compacts the current heap zone and returns the size of the largest contiguous block available for temporary allocation.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for TempMaxMem are

Trap macro	Selector
_OSDispatch	\$0015

SPECIAL CONSIDERATIONS

Because TempMaxMem could move memory, you should not call it at interrupt time.

Accessing Heap Zones

The majority of applications, which allocate memory in their application heap zone only, do not need to use any of the routines in this section. The few applications that do allocate memory in zones other than the application heap zone can use the GetZone function and the SetZone procedure to get and set the current zone, the ApplicationZone and SystemZone functions to obtain pointers to the application and system zones, and the HandleZone and PtrZone functions to find the zones in which relocatable and nonrelocatable blocks lie.

GetZone

To find which zone is current, use the GetZone function.

```
FUNCTION GetZone: THz;
```

DESCRIPTION

The GetZone function returns a pointer to the current heap zone.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for GetZone are

Registers on exit

A0	Pointer to current heap zone
D0	Result code

The global variable TheZone contains a pointer to the current heap zone.

RESULT CODES

noErr	0	No error
-------	---	----------

SetZone

To change the current heap zone, you can use the `SetZone` procedure.

```
PROCEDURE SetZone (hz: THz);
```

`hz` A pointer to the heap zone to make current.

DESCRIPTION

The `SetZone` procedure makes the zone to which `hz` points the current heap zone. Often, you use the `SetZone` procedure in conjunction with one of the `ApplicationZone`, `SystemZone`, `HandleZone`, and `PtrZone` functions. For example, the code `SetZone(SystemZone)` makes the system heap zone current.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `SetZone` are

Registers on entry

`A0` Pointer to new current heap zone

Registers on exit

`D0` Result code

RESULT CODES

`noErr` 0 No error

ApplicationZone

To obtain a pointer to the application heap zone, you can use the `ApplicationZone` function.

```
FUNCTION ApplicationZone: THz;
```

DESCRIPTION

The `ApplicationZone` function returns a pointer to the original application heap zone.

ASSEMBLY-LANGUAGE INFORMATION

The global variable `ApplZone` contains a pointer to the original application heap zone.

SystemZone

To obtain a pointer to the system heap zone, you can use the `SystemZone` function.

```
FUNCTION SystemZone: THz;
```

DESCRIPTION

The `SystemZone` function returns a pointer to the system heap zone.

ASSEMBLY-LANGUAGE INFORMATION

The global variable `SysZone` contains a pointer to the system heap zone.

HandleZone

If you need to know which heap zone contains a particular relocatable block, you can use the `HandleZone` function.

```
FUNCTION HandleZone (h: Handle): THz;
```

`h` A handle to a relocatable block.

DESCRIPTION

The `HandleZone` function returns a pointer to the heap zone containing the relocatable block whose handle is `h`. In case of an error, the result returned by `HandleZone` is undefined and should be ignored.

IMPORTANT

If the handle `h` is empty (that is, if it points to a `NIL` master pointer), `HandleZone` returns a pointer to the heap zone that contains the master pointer. ▲

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HandleZone` are

Registers on entry

A0 Handle whose zone is to be found

Registers on exit

A0 Pointer to handle's heap zone

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>memWZErr</code>	-111	Attempt to operate on a free block

PtrZone

If you have allocated a nonrelocatable block and need to know in which zone it lies, you can use the `PtrZone` function.

```
FUNCTION PtrZone (p: Ptr): THz;
```

`p` A pointer to a nonrelocatable block.

DESCRIPTION

The `PtrZone` function returns a pointer to the heap zone containing the nonrelocatable block pointed to by `p`.

In case of an error, the result returned by `PtrZone` is undefined and should be ignored.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `PtrZone` are

Registers on entry

A0 Pointer whose zone is to be found

Registers on exit

A0 Pointer to heap zone of nonrelocatable block

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>memWZErr</code>	-111	Attempt to operate on a free block

Manipulating Heap Zones

The Memory Manager provides several routines for initializing and resizing heap zones.

To obtain information about the current application partition, applications can call the `GetApplLimit` function and the `TopMem` function. If your application uses the stack extensively, you might want to ensure that the stack is set to at least some minimum size, at the expense of the heap. To do so, use the `SetApplLimit` procedure to change the application heap limit before you call the `MaxApplZone` procedure.

To initialize a new heap zone, use the `InitZone` procedure. The Operating System automatically initializes the application zone by calling the `SetApplBase` procedure, which subsequently calls the `InitApplZone` procedure.

GetApplLimit

Use the `GetApplLimit` function to get the application heap limit, beyond which the application heap cannot expand.

```
FUNCTION GetApplLimit: Ptr;
```

DESCRIPTION

The `GetApplLimit` function returns the current application heap limit. The Memory Manager expands the application heap only up to the byte preceding this limit.

Nothing prevents the stack from growing below the application limit. If the Operating System detects that the stack has crashed into the heap, it generates a system error. To avoid this, use `GetApplLimit` and the `SetApplLimit` procedure to set the application limit low enough so that a growing stack does not encounter the heap.

Note

The `GetApplLimit` function does not indicate the amount of memory available to your application. ♦

ASSEMBLY-LANGUAGE INFORMATION

The global variable `ApplLimit` contains the current application heap limit.

SetApplLimit

Use the `SetApplLimit` procedure to set the application heap limit, beyond which the application heap cannot expand.

```
PROCEDURE SetApplLimit (zoneLimit: Ptr);
```

zoneLimit A pointer to a byte in memory demarcating the upper boundary of the application heap zone. The zone can grow to include the byte preceding `zoneLimit` in memory, but no further.

DESCRIPTION

The `SetApplLimit` procedure sets the current application heap limit to `zoneLimit`. The Memory Manager then can expand the application heap only up to the byte

Memory Manager

preceding the application limit. If the zone already extends beyond the specified limit, the Memory Manager does not cut it back but does prevent it from growing further.

Note

The `zoneLimit` parameter is not a byte count, but an absolute byte in memory. Thus, you should use the `SetApplLimit` procedure only with a value obtained from the Memory Manager functions `GetApplLimit` or `ApplicationZone`. ♦

You cannot change the limit of zones other than the application heap zone.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `SetApplLimit` are

Registers on entry

A0 Pointer to desired new zone limit

Registers on exit

D0 Result code

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory

TopMem

To find out the location of the top of an application's partition, you can use the `TopMem` function, which exhibits special behavior during the startup process.

```
FUNCTION TopMem: Ptr;
```

DESCRIPTION

Except during the startup process, the `TopMem` function returns a pointer to the byte at the top of an application's partition, directly above the jump table. The function does this to maintain compatibility with programs that check `TopMem` to find out how much memory is installed in a computer. To obtain this information, you can currently use the `Gestalt` function.

The function exhibits special behavior at startup time, and the value it returns controls the amount by which an extension can lower the value of the global variable `BufPtr` at startup time. If you are writing a system extension, you should not lower the value of `BufPtr` by more than `MemTop DIV 2 + 1024`. If you do lower `BufPtr` too far, the startup process generates an out-of-memory system error.

You should never need to call `TopMem` except during the startup process.

ASSEMBLY-LANGUAGE INFORMATION

The `TopMem` function returns the value of the `MemTop` global variable.

InitZone

If you want to use heap zones other than the original application heap zone, a temporary memory zone, or the system heap zone, you can use the `InitZone` procedure to initialize a new heap zone.

```
PROCEDURE InitZone (pGrowZone: ProcPtr; cMoreMasters: Integer;
                   limitPtr, startPtr: Ptr);
```

pGrowZone A pointer to a grow-zone function for the new heap zone. If you do not want the new zone to have a grow-zone function, set this parameter to `NIL`.

cMoreMasters The number of master pointers that should be allocated at a time for the new zone. The Memory Manager allocates this number initially, and, if it needs to allocate more later, allocates them in increments of this same number.

limitPtr The first byte beyond the end of the zone.

startPtr The first byte of the new zone.

DESCRIPTION

The `InitZone` procedure creates a new heap zone, initializes its header and trailer, and makes it the current zone. Although the new zone occupies memory addresses from `startPtr` through `limitPtr-1`, the new zone includes a zone header and a zone trailer. In addition, the new zone contains a block header for the master pointer block and 4 bytes for each master pointer. If you need to create a zone with some specific number of usable bytes, see “Organization of Memory,” beginning on page 6-331, for details on the sizes of the zone header, zone trailer, and block header.

Note

The sizes of zones and block headers may change in future system software versions. You should ensure that your zones are large enough to accommodate a reasonable increase in the sizes of those structures. ♦

SPECIAL CONSIDERATIONS

Because `InitZone` changes the current zone, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `InitZone` are

Registers on entry

A0 Pointer to parameter block

Registers on exit

D0 Result code

The parameter block whose address is passed in register A0 has no Pascal type definition. It has this structure:

Parameter block

→	<code>startPtr</code>	<code>Ptr</code>	The first byte of the new zone.
→	<code>limitPtr</code>	<code>Ptr</code>	The first byte beyond the new zone.
→	<code>cMoreMasters</code>	<code>Integer</code>	The number of master pointers to be allocated at a time.
→	<code>pGrowZone</code>	<code>ProcPtr</code>	A pointer to the new zone's grow-zone function, or <code>NIL</code> if none.

RESULT CODES

`noErr` 0 No error

InitApplZone

The Process Manager calls the `InitApplZone` procedure indirectly when it starts up your application. You should never need to call it. It is documented for completeness only.

```
PROCEDURE InitApplZone;
```

DESCRIPTION

The `InitApplZone` procedure initializes the application heap zone and makes it the current zone. The Memory Manager discards the contents of any previous application zone and discards all previously existing blocks in that zone. The procedure sets the zone's grow-zone function to `NIL`.

▲ **WARNING**

Reinitializing the application zone from within a running program is dangerous, because the application's code itself normally resides in the application zone. To do so safely, you must make sure that the code containing the `InitApplZone` call is not in the application zone. ▲

SPECIAL CONSIDERATIONS

You should not call `InitApplZone` at all, but, if you must, be sure not to call it at interrupt time because it could purge and allocate memory.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `InitApplZone` are

Registers on exit

D0 Result code

RESULT CODES

`noErr` 0 No error

SetApplBase

The Process Manager calls the `SetApplBase` procedure when it starts up your application. You should never need to call it. It is documented for completeness only.

```
PROCEDURE SetApplBase (startPtr: Ptr);
```

`startPtr` The starting address for the application heap zone to be initialized.

DESCRIPTION

The `SetApplBase` procedure sets the starting address of the application heap zone for the application being initialized to the address designated by `startPtr`, and then calls the `InitApplZone` procedure.

▲ WARNING

Like `InitApplZone`, `SetApplBase` is a potentially dangerous operation, because the program's code itself normally resides in the application heap zone. To do so safely, you must make sure that the code containing the `SetApplBase` call is not in the application zone. ▲

SPECIAL CONSIDERATIONS

You should not call `SetApplBase` at all, but, if you must, be sure not to call it at interrupt time because it affects memory.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `SetApplBase` are

Registers on exit

D0 Result code

RESULT CODES

`noErr` 0 No error

Application-Defined Routines

The Memory Manager provides a means for you to intervene in its otherwise automatic operations by allowing you to define a grow-zone function and a purge-warning procedure.

Note

Many applications use a grow-zone function as part of a general strategy to prevent low-memory situations. Most applications, however, do not need to use purge-warning procedures. ♦

Grow-Zone Functions

The Memory Manager calls your application's grow-zone function whenever it cannot find enough contiguous memory to satisfy a memory allocation request and has exhausted other means of obtaining the space.

MyGrowZone

A grow-zone function should have the following form:

```
FUNCTION MyGrowZone (cbNeeded: Size): LongInt;
```

`cbNeeded` The physical size, in bytes, of the needed block, including the block header. The grow-zone function should attempt to create a free block of at least this size.

DESCRIPTION

Whenever the Memory Manager has exhausted all available means of creating space within your application heap—including purging, compacting, and (if possible) expanding the heap—it calls your application-defined grow-zone function. The grow-zone function can do whatever is necessary to create free space in the heap. Typically, a grow-zone function marks some unneeded blocks as purgeable or releases an emergency memory reserve maintained by your application.

The grow-zone function should return a nonzero value equal to the number of bytes of memory it has freed, or zero if it is unable to free any. When the function returns a nonzero value, the Memory Manager once again purges and compacts the heap zone and tries to reallocate memory. If there is still insufficient memory, the Memory Manager calls the grow-zone function again (but only if the function returned a nonzero value the previous time it was called). This mechanism allows your grow-zone function to release just a little bit of memory at a time. If the amount it releases at any time is not enough, the Memory Manager calls it again and gives it the opportunity to take more drastic measures.

The Memory Manager might designate a particular relocatable block in the heap as protected; your grow-zone function should not move or purge that block. You can determine which block, if any, the Memory Manager has protected by calling the `GZSaveHnd` function in your grow-zone function.

Remember that a grow-zone function is called while the Memory Manager is attempting to allocate memory. As a result, your grow-zone function should not allocate memory itself or perform any other actions that might indirectly cause memory to be allocated (such as calling routines in unloaded code segments or displaying dialog boxes).

You install a grow-zone function by passing its address to the `InitZone` procedure when you create a new heap zone or by calling the `SetGrowZone` procedure at any other time.

SPECIAL CONSIDERATIONS

Your grow-zone function might be called at a time when the system is attempting to allocate memory and the value in the A5 register is not correct. If your function accesses your application's A5 world or makes any trap calls, you need to set up and later restore the A5 register by calling `SetCurrentA5` and `SetA5`. See the chapter "Memory Management Utilities" in this book for a description of these two functions.

Because of the optimizations performed by some compilers, the actual work of the grow-zone function and the setting and restoring of the A5 register might have to be placed in separate procedures.

SEE ALSO

See the chapter "Introduction to Memory Management" in this book for a definition of a sample grow-zone function.

Purge-Warning Procedures

The Memory Manager calls your application's purge-warning procedure whenever it is about to purge a relocatable block from your application heap.

MyPurgeProc

A purge-warning procedure should have the following form:

```
PROCEDURE MyPurgeProc (h: Handle);
```

h A handle to the block that is about to be purged.

DESCRIPTION

Whenever the Memory Manager needs to purge a block from the application heap, it first calls any application-defined purge-warning procedure that you have installed. The purge-warning procedure can, if necessary, save the contents of that block or otherwise respond to the warning.

Your purge-warning procedure is called during a memory-allocation request. As a result, you should not call any routines that might cause memory to be moved or purged. In particular, if you save the data of the block in a file, the file should already be open when your purge-warning procedure is called, and you should write the data synchronously.

You should not dispose of or change the purgeable status of the block whose handle is passed to your procedure.

To install a purge-warning procedure, you need to assign its address to the `purgeProc` field of the associated zone header.

Note

If you call the Resource Manager procedure `SetResPurge` with the parameter `TRUE`, any existing purge-warning procedure is replaced by a purge-warning procedure installed by the Resource Manager. You can execute both warning procedures by calling `SetResPurge`, saving the existing value of the `purgeProc` field of the zone header, and then reinstalling your purge-warning procedure. Your purge-warning procedure should call the Resource Manager's purge-warning procedure internally. ♦

SPECIAL CONSIDERATIONS

Your purge-warning procedure might be called at a time when the system is attempting to allocate memory and the value in the A5 register is not correct. If your function accesses your application's A5 world or makes any trap calls, you need to set up and later restore the A5 register by calling `SetCurrentA5` and `SetA5`.

Because of the optimizations performed by some compilers, the actual work of the purge-warning procedure and the setting and restoring of the A5 register might have to be placed in separate procedures.

Your purge-warning procedure is called for *every* handle that is about to be purged (not necessarily for every purgeable handle in your heap, however). Your procedure should be able to determine quickly whether the handle it is passed is one whose associated data needs to be saved or otherwise processed.

SEE ALSO

See “Installing a Purge-Warning Procedure” on page 6-329 for a definition of a sample purge-warning procedure and for instructions on installing the procedure.

Result Codes

noErr	0	No error
paramErr	-50	Error in parameter list
memROZErr	-99	Operation on a read-only zone
memFullErr	-108	Not enough memory
nilHandleErr	-109	NIL master pointer
memWZErr	-111	Attempt to operate on a free block
memPurErr	-112	Attempt to purge a locked block
memBCErr	-115	Block check failed
memLockedErr	-117	Block is locked

Event Manager

This chapter describes the constants, data types, and functions for the Event Manager and Operating System Event Manager. It also describes the 'SIZE' resource.

Event Manager Constants and Types

This section describes the enumerators and types for event kinds, event masks, event modifier flags, and posting options; the event record structure, target ID structure, high-level event message structure, and structure of the Operating System event queue; and the filter function types and macros.

The Event Manager uses event records to return information about events. You can use a target ID structure to specify or identify the address of another application or process with which your application is communicating. If your application supplies a filter function as a parameter to the `GetSpecificHighLevelEvent` function (page 7-431), your filter function receives information about high-level events in a high-level event message structure.

Event Kinds

Enumerators of type `EventKind` specify the kind of event described by an event record or posted with `PPostEvent` (page 7-446) or `PostEvent` (page 7-448).

```
typedef UInt16 EventKind;    /* event kinds */
enum {
    /* event codes */
    nullEvent                = 0,        /* no other pending events */
    mouseDown                = 1,        /* mouse button pressed */
    mouseUp                  = 2,        /* mouse button released */
    keyDown                  = 3,        /* key pressed */
}
```

Event Manager

```

keyUp                = 4,          /* key released */
autoKey              = 5,          /* key repeatedly held down */
updateEvt            = 6,          /* window needs updating */
diskEvt              = 7,          /* disk inserted */
activateEvt          = 8,          /* activate/deactivate window */
osEvt                = 15         /* operating-system event
                                /* (suspend, resume, or
                                /* mouse-moved) */

kHighLevelEvent      = 23         /* high-level event */
/*message codes for operating-system events */
mouseMovedMessage     = 0x00FA,    /* mouse-moved event */
suspendResumeMessage  = 0x0001,    /* suspend or resume event */
/*flags for suspend and resume events */
resumeFlag            = 1,         /* resume event */
convertClipboardFlag  = 2,         /* Clipboard conversion
                                /* required */
};

```

Enumerator descriptions

<code>nullEvent</code>	The event code indicating that there are no other pending events.
<code>mouseDown</code>	The event code indicating that the mouse button has been pressed.
<code>mouseUp</code>	The event code indicating that the mouse button has been released.
<code>keyDown</code>	The event code indicating that a key has been pressed.
<code>keyUp</code>	The event code indicating that a key has been released.
<code>autoKey</code>	The event code indicating that a key has been repeatedly held down.
<code>updateEvt</code>	The event code indicating that a window needs updating.
<code>diskEvt</code>	The event code indicating that a disk has been inserted.
<code>activateEvt</code>	The event code indicating that a window has been activated or deactivated.
<code>osEvt</code>	The event code indicating a suspend, resume, or mouse-moved operating-system event.
<code>kHighLevelEvent</code>	A high-level event.
<code>mouseMovedMessage</code>	The message code indicating the mouse-moved operating-system event.

Event Manager

suspendResumeMessage

The message code indicating a suspend or resume operating-system event.

resumeFlag

Flag for a resume event.

convertClipboardFlag

Flag indicating that Clipboard conversion is required for a suspend or resume event.

Note that in System 7, event kinds with the values 9 through 14 are undefined and reserved for future use by Apple. All other values for the `what` field are also reserved for use by Apple.

For information about the use of these enumerators in the event record, see “The Event Record” (page 7-414).

Event Masks

Enumerators of type `EventMask` specify what kind of events you want your application to receive. You use these enumerators with these Event Manager functions: `GetNextEvent` (page 7-428), `WaitNextEvent` (page 7-423), `EventAvail` (page 7-427), `OSEventAvail` (page 7-439), `FlushEvents` (page 7-433), `GetOSEvent` (page 7-438). To set the system event mask, which determines which low-level events your application receives, you use `SetEventMask` (page 7-441). For information about all these functions, see “Receiving Events” (page 7-422).

```
typedef UInt16 EventMask;    /* event mask */
enum {
    mDownMask      = 0x0002,    /* mouse-down event (bit 1) */
    mUpMask        = 0x0004,    /* mouse-up event (bit 2) */
    keyDownMask    = 0x0008,    /* key-down event (bit 3) */
    keyUpMask      = 0x0010,    /* key-up event (bit 4) */
    autoKeyMask    = 0x0020,    /* auto-key event (bit 5) */
    updateMask     = 0x0040,    /* update event (bit 6) */
    diskMask       = 0x0080,    /* disk-inserted event (bit 7) */
    activMask      = 0x0100,    /* activate event (bit 8) */
    highLevelEventMask = 0x0400, /* high-level event (bit 10) */
    osMask         = 0x0800,    /* operating-system event (bit 15) */
    everyEvent     = 0xFFFF,    /* every event */
    /* event message masks for keyboard events */
    charCodeMask   = 0x000000FF, /* use to get character code */
    keyCodeMask    = 0x0000FF00, /* use to get key code */
}
```

Event Manager

```
adbAddrMask      = 0x00FF0000,    /* ADB address if ADB keyboard */
osEvtMessageMask = 0xFF000000L    /* can use to extract message code */
};
```

Enumerator descriptions

<code>mDownMask</code>	The enumerator indicating you want your application to receive a mouse-down event.
<code>mUpMask</code>	The enumerator indicating you want your application to receive a mouse-up event.
<code>keyDownMask</code>	The enumerator indicating you want your application to receive a key-down event.
<code>keyUpMask</code>	The enumerator indicating you want your application to receive a key-up event.
<code>autoKeyMask</code>	The enumerator indicating you want your application to receive an auto-key event.
<code>updateMask</code>	The enumerator indicating you want your application to receive an update event.
<code>diskMask</code>	The enumerator indicating you want your application to receive a disk-inserted event.
<code>activMask</code>	The enumerator indicating you want your application to receive an activate event.
<code>highLevelEventMask</code>	The enumerator indicating you want your application to receive a high-level event.
<code>osMask</code>	The enumerator indicating you want your application to receive an operating-system event
<code>everyEvent</code>	The enumerator indicating you want your application to receive every event.
<code>charCodeMask</code>	The enumerator indicating you want your application to receive a character-code keyboard event.
<code>keyCodeMask</code>	The enumerator indicating you want your application to receive a key-code keyboard event.
<code>adbAddrMask</code>	The enumerator indicating you want your application to receive an ADB address if there is an ADB keyboard.
<code>osEvtMessageMask</code>	The enumerator indicating you want your application to receive a keyboard event that can be used to extract a message code.

Event Modifier Flags

Enumerators of type `EventModifiers` are used in the event record and its equivalent `EvQE1` structure in the event queue to specify the state of the modifier keys and the mouse button at the time the event was posted.

```
typedef UInt16 EventModifiers; /* event modifiers */
enum {
    activeFlag      = 0x0001, /* set if window being activated or if
                               /* mouse-down event caused foreground
                               /* switch */
    btnState        = 0x0080, /* set if mouse button up */
    cmdKey          = 0x0100, /* set if Command key down */
    shiftKey        = 0x0200, /* set if Shift key down */
    alphaLock       = 0x0400, /* set if Caps Lock key down */
    optionKey       = 0x0800, /* set if Option key down */
    controlKey      = 0x1000  /* set if Control key down */
};
```

Enumerator descriptions

<code>activeFlag</code>	The enumerator that indicates a window is being activated or that a mouse-down event caused a foreground switch.
<code>btnState</code>	The enumerator indicating that the mouse button has been released.
<code>cmdKey</code>	The enumerator indicating that the Command key is being pressed.
<code>shiftKey</code>	The enumerator indicating that the Shift key is being pressed.
<code>alphaLock</code>	The enumerator indicating that the Caps Lock key is being pressed.
<code>optionKey</code>	The enumerator indicating that the Option key is being pressed.
<code>controlKey</code>	The enumerator indicating that the Control key is being pressed.

For information about the event record and the `EvQE1` structure, see “The Event Record” (page 7-414) and “The Event Queue” (page 7-419).

Posting Options

You use posting option enumerators with `PostHighLevelEvent` (page 7-443) to indicate how you are specifying the receiver of the high-level event and how you want the event to be delivered.

```
enum {                                /* posting option enumerators */
    receiverIDMask                    = 0x0000F000,      /* mask for receiver ID bits */
    receiverIDisPSN                   = 0x00008000,      /* ID is process serial number */
    receiverIDisSignature              = 0x00007000,      /* ID is creator signature */
    receiverIDisSessionID              = 0x00006000,      /* ID is PPC session reference
                                                             /* number */
    receiverIDisTargetID               = 0x00005000,      /* ID is port name and location name */
    systemOptionsMask                  = 0x0000F00,       /* system options */
    nReturnReceipt                     = 0x00000200,      /* return receipt requested */
    priorityMask                       = 0x000000FF,       /* priority */
    nAttnMsg                           = 0x00000001       /* give this message priority */

    /* class and ID values for return receipt */
    HighLevelEventMsgClass              = 'jaym',          /* high-level events message class */
    rtnReceiptMsgID                     = 'rtrn',          /* return receipt message ID */

    /* modifiers values in return receipt */
    msgWasPartiallyAccepted             = 2,              /* message was partially accepted */
    msgWasFullyAccepted                 = 1,              /* message was fully accepted */
    msgWasNotAccepted                   = 0                /* message was not accepted */
};
```

Enumerator descriptions

<code>receiverIDMask</code>	The posting enumerator indicating that the receiver ID consists of the bits for the event to be delivered.
<code>receiverIDisPSN</code>	The posting enumerator indicating that the receiver ID is the process serial number for the event to be delivered.
<code>receiverIDisSignature</code>	The posting enumerator indicating that the receiver ID is a creator signature.
<code>receiverIDisSessionID</code>	The posting enumerator indicating that the receiver ID is a PPC session reference number.

Event Manager

<code>receiverIDIsTargetID</code>	The posting enumerator indicating that the receiver ID is a port name and location name.
<code>systemOptionsMask</code>	The posting enumerator indicating that the system options mask.
<code>nReturnReceipt</code>	The posting enumerator indicating that a return receipt has been requested.
<code>priorityMask</code>	The posting enumerator indicating priority.
<code>nAttnMsg</code>	The posting enumerator indicating that this message should be given priority.
<code>HighLevelEventMsgClass</code>	The enumerator indicating a high-level event message class for return receipt.
<code>rttnReceiptMsgID</code>	The posting enumerator indicating the return receipt message ID.
<code>msgWasPartiallyAccepted</code>	The posting enumerator value in the return receipt that indicates the message was partially accepted.
<code>msgWasFullyAccepted</code>	The posting enumerator value in the return receipt that indicates the message was fully accepted.
<code>msgWasNotAccepted</code>	The posting enumerator value in the return receipt that indicates the message was fully accepted.

The Key Map Type

The type `KeyMap` is used in the `GetKeys` function (page 7-455) to return the current state of the keyboard, including the keypad, if any. The `KeyMap` type is interpreted as an array of 128 elements, each having a Boolean value. Each key on the keyboard or keypad corresponds to an element in the `KeyMap` array. The index for a particular key is the same as the key's virtual key code minus 1. For example, the key with virtual key code 38 (the "J" key on the Apple Keyboard II) is the 38th element in the returned array. A `KeyMap` element is `true` if the corresponding key is down and `false` if it isn't. The maximum number of keys that can be down simultaneously is two character keys plus any combination of the five modifier keys.

```
typedef long KeyMap[4];      /* records state of keyboard */
```

The Event Record

When your application uses an Event Manager function to retrieve an event, the Event Manager returns information about the retrieved event in an event record structure, which is a structure of type `EventRecord`.

```
struct EventRecord {
    EventKind      what;          /* event code */
    UInt32         message;       /* event message */
    UInt32         when;          /* ticks since startup */
    Point          where;         /* mouse location */
    EventModifiers modifiers;     /* modifier flags */
};
typedef struct EventRecord EventRecord;
```

Field descriptions

what	The kind of event received. The Event Manager specifies the kind of event with one of the values defined by the <code>EventKind</code> enumeration (page 7-407).
message	Additional information associated with the event. The interpretation of this information depends on the event

type. The contents of the `message` field for each event type are summarized here:

Event type	Event message
null, mouse-up, mouse-down	Undefined.
key-up, key-down, auto-key	The low-order word contains the character code and virtual key code, which you can access with the constants <code>charCodeMask</code> and <code>keyCodeMask</code> , respectively. For Apple Desktop Bus (ADB) keyboards, the low byte of the high-order word contains the ADB address of the keyboard where the keyboard event occurred. The high byte of the high-order word is reserved.
update, activate	Pointer to the window to update, activate, or deactivate.
disk-inserted	Drive number in low-order word, File Manager result code in high-order word.
resume	The <code>suspendResumeMessage</code> enumerator in bits 24–31 and a 1 (the <code>resumeFlag</code> enumerator) in bit 0 indicate the event is a resume event. Bit 1 contains a 1 (the <code>convertClipboardFlag</code> enumerator) if Clipboard conversion is required, and bits 2–23 are reserved.

	Event type	Event message
	suspend	The <code>suspendResumeMessage</code> enumerator in bits 24–31 and a 0 in bit 0 to indicate the event is a suspend event. Bit 1 is undefined, and bits 2–23 are reserved.
	mouse-moved	The <code>mouseMovedMessage</code> enumerator in bits 24–31. Bits 2–23 are reserved, and bit 0 and bit 1 are undefined.
	high-level	Class of events to which the high-level event belongs. The <code>message</code> and <code>where</code> fields of a high-level event define the specific type of high-level event received.
when	The <code>when</code> field indicates the time when the event was posted (in ticks since system startup).	
where	<p>For low-level events and operating-system events, the <code>where</code> field contains the location of the cursor at the time the event was posted (in global coordinates).</p> <p>For high-level events, the <code>where</code> field contains a second event specifier, the event ID. The event ID defines the particular type of event within the class of events defined by the <code>message</code> field of the high-level event. For high-level events, you should interpret the <code>where</code> field as having the data type <code>OSType</code>, not <code>Point</code>.</p>	
modifiers	<p>The <code>modifiers</code> field contains information about the state of the modifier keys and the mouse button at the time the event was posted. For activate events, this field also indicates whether the window should be activated or deactivated. In System 7 it also indicates whether the mouse-down event caused your application to switch to the foreground.</p> <p>Each of the modifier keys is represented by a specific bit in the <code>modifiers</code> field of the event record structure. The modifier keys include the Option, Command, Caps Lock, Control, and Shift keys. If your application attaches special meaning to any of these keys in combination with other keys or when the mouse button is down, you can test the</p>	

state of the `modifiers` field to determine the action your application should take. For example, you can use this information to determine whether the user pressed the Command key and another key to make a menu choice.

The Target ID Structure

When you send a high-level event to another application, you can use a target ID structure to specify the recipient of the event. When you receive a high-level event, the `AcceptHighLevelEvent` function (page 7-429) uses a target ID structure to return information about the sender of the event.

A target ID structure is defined by a structure of type `TargetID`.

```
struct TargetID {
    long          sessionID;      /* session reference number */
    PPCPortRec    name;          /* port name */
    LocationNameRec location;     /* location name */
    PPCPortRec    recvrName;     /* reserved */
};

typedef struct TargetID TargetID;
typedef TargetID *TargetIDPtr, **TargetIDHandle, **TargetIDHdl;
```

Field descriptions

<code>sessionID</code>	For high-level events that your application receives, this field contains the session reference number created by the PPC Toolbox. This is a 32-bit number that uniquely identifies a PPC Toolbox session (or connection) with another application. This field is not used by your application when sending a high-level event to another process. (To send a high-level event that specifies the recipient by session reference number, provide a pointer to a session reference number in the <code>receiverID</code> parameter and use the <code>receiverIDIsSessionID</code> constant in the <code>postingOptions</code> parameter to <code>PostHighLevelEvent</code> (page 7-443).)
<code>name</code>	For high-level events that your application receives, this field contains a PPC port record that specifies the port name of the process from which the high-level event originated. When sending a high-level event to a process on a local or remote computer, you can specify the port

	name of the recipient process in a PPC port record that you provide in this field.
	If the sending application is on the same computer as the receiving application, you can determine the sending application's process serial number by calling the <code>GetProcessSerialNumberFromPortName</code> function (page 7-450).
location	For high-level events that your application receives, this field contains a location name record that identifies the location name of the process from which the high-level event originated. When sending a high-level event to a process on a local or remote computer, you can specify the location name of the recipient process in a location name record that you provide in this field.
recvrName	This field is reserved.

The High-Level Event Message Structure

You can search your application's high-level event queue for a specific high-level event by using the `GetSpecificHighLevelEvent` function (page 7-431) and providing a filter function. Your filter function receives a pointer to a high-level event message structure that contains information about a high-level event.

For information on getting a function descriptor for your filter function, see "Filter Function Pointer and Macro" (page 7-420). For information on how to define a filter function, see "Filter Function for Searching the High-Level Event Queue" (page 7-467).

A structure of type `HighLevelEventMsg` defines a high-level event message structure.

```
struct HighLevelEventMsg {                                /* high-level event message structure */
    unsigned short    HighLevelEventMsgHeaderLength;    /* reserved */
    unsigned short    version;                          /* reserved */
    unsigned long     reserved1;                        /* reserved */
    EventRecord       theMsgEvent;                      /* event record of
                                                         /* high-level event */
    unsigned long     userRefCon;                       /* user reference
                                                         /* constant */
    unsigned long     postingOptions;                   /* reserved */
    unsigned long     msgLength;                       /* reserved */};
```

Event Manager

```
typedef struct HighLevelEventMsg HighLevelEventMsg;
typedef HighLevelEventMsg *HighLevelEventMsgPtr,
                        **HighLevelEventMsgHandle,
                        **HighLevelEventMsgHdl;
```

Field descriptions

<code>HighLevelEventMsgHeaderLength</code>	Reserved for use by the Event Manager.
<code>version</code>	Reserved for use by the Event Manager.
<code>reserved1</code>	Reserved for use by the Event Manager.
<code>theMsgEvent</code>	The event record of a high-level event. Your filter function can compare the fields of this event record to determine whether the high-level event is the desired event. If your filter function finds the desired event, it should call <code>AcceptHighLevelEvent</code> (page 7-429) to accept the event and remove the event from the high-level event queue, and return <code>true</code> as its function result.
<code>userRefCon</code>	A unique number that identifies the communication associated with this event.
<code>postingOptions</code>	Reserved for use by the Event Manager.
<code>msgLength</code>	Reserved for use by the Event Manager.

The Event Queue

The event queue is a standard Macintosh Operating System queue that the Operating System Event Manager maintains. Only mouse-up, mouse-down, key-up, key-down, auto-key, and disk-inserted events are stored in the Operating System event queue. In most cases, your application should not access the event queue directly. Instead you usually use the `WaitNextEvent` function (page 7-423), which can retrieve events from this queue as well as from other sources.

The event queue consists of a header followed by the actual entries in the queue. The event queue has the same header as all standard Macintosh Operating System queues. The `QHdr` structure defines the queue header.

Event Manager

```

struct QHdr {
    short      qFlags;          /* queue flags */
    QElemPtr   qHead;          /* first queue entry */
    QElemPtr   qTail;          /* last queue entry */
};

```

A structure of type `EvQEl` defines an entry in the Operating System event queue.

```

struct EvQEl {      /* Operating System event queue */
    QElemPtr   qLink;          /* next queue entry */
    short      qType;          /* queue type (evType) */
    EventKind   evtQWhat;      /* event code */
    UInt32     evtQMessage;     /* event message */
    UInt32     evtQWhen;       /* ticks since startup */
    Point       evtQWhere;      /* mouse location */
    EventModifiers evtQModifiers; /* modifier flags */
};
typedef struct EvQEl EvQEl;
typedef EvQEl *EvQElPtr;

```

Each entry in the event queue begins with 4 bytes of flags followed by a pointer to the next queue entry. The flags are maintained by and internal to the Operating System Event Manager. The queue entries are linked by pointers, and the first field of the `EvQEl` data type, which represents the structure of a queue entry, begins with a pointer to the next queue entry. Thus, you cannot directly access the flags using the `EvQEl` data type.

Filter Function Pointer and Macro

The `GetSpecificHighLevelEvent` function (page 7-431) takes a universal procedure pointer to an application-defined filter function used to search for a specific event. The Event Manager declares the type for an application-defined specific event filter function as follows:

```

typedef pascal Boolean (*GetSpecificFilterProcPtr)
    (void *contextPtr,
     HighLevelEventMsgPtr msgBuff,
     const TargetID *sender);

```

Event Manager

For information about writing a specific filter function, see “Filter Function for Searching the High-Level Event Queue” (page 7-467).

The Event Manager defines the universal procedure pointer `GetSpecificFilterUPP` for an application-defined filter function.

```
typedef UniversalProcPtr GetSpecificFilterUPP;
```

The Event Manager also defines the macro `NewGetSpecificFilterProc` for obtaining a `GetSpecificFilterUPP`:

```
#define NewGetSpecificFilterProc(userRoutine)\
    NewRoutineDescriptor((ProcPtr)(userRoutine),\
        uppGetSpecificFilterProcInfo, GetCurrentArchitecture())
```

You typically use the `NewGetSpecificFilterProc` macro like this:

```
GetSpecificFilterUPP myFilterFunctionProc;\
myFilterFunctionProc = NewGetSpecificFilterProc(MyFilter);
```

The Event Manager also defines the macro `CallGetSpecificFilterProc` for calling a `GetSpecificFilterUPP`. You normally don’t need to call this macro directly:

```
#define CallGetSpecificFilterProc(userRoutine, contextPtr,\
    msgBuff, sender)\
    CallUniversalProc((UniversalProcPtr)(userRoutine),\
        uppGetSpecificFilterProcInfo, (contextPtr), (msgBuff), (sender))
```

For more information about universal procedure pointers and the Mixed Mode Manager, see *Inside Macintosh: PowerPC System Software*.

Event Manager Functions

The Event Manager includes functions for receiving events, receiving and sending high-level events, and searching for specific high-level events. The Event Manager also provides functions for converting between process serial numbers and port names, getting information about the state of the mouse button, reading the keyboard, and getting timing information.

Receiving Events

You can use the `WaitNextEvent` (page 7-423) or `GetNextEvent` (page 7-428) function to retrieve an event from the Event Manager and remove the event from the event stream. To provide greater support for multitasking, however, you should use the `WaitNextEvent` function instead of `GetNextEvent` whenever possible. You can use the `EventAvail` function (page 7-427) to look at an event without removing it from the event stream. You can use the `AcceptHighLevelEvent` function (page 7-429) to get additional information associated with a high-level event and `GetSpecificHighLevelEvent` (page 7-431) to search for a specific high-level event.

The `FlushEvents` (page 7-433) function removes all low-level events from the Operating System event queue. In general, your application should not empty the event queue.

You can use the `SystemClick` function (page 7-435) to route events to desk accessories when necessary. The `SystemTask` (page 7-436) and `SystemEvent` (page 7-437) functions are used by the Event Manager, and your application usually does not need to call these two functions.

You usually use the functions provided by the Toolbox Event Manager to retrieve events from the event stream. Even if you are interested only in the events stored in the Operating System event queue, you can retrieve these events using the Toolbox Event Manager by setting the event mask to mask out all events except keyboard, mouse, and disk-inserted events. However, you can choose to use Operating System Event Manager functions to perform this task.

The Operating System Event Manager provides two functions, `GetOSEvent` (page 7-438) and `OSEventAvail` (page 7-439), to retrieve events from the Operating System event queue. In most cases, your application will not need to use these two functions.

If your application needs to receive key-up events, you can change the system event mask of your application using the `SetEventMask` function (page 7-441). The `GetEvQHdr` function (page 7-442) returns a pointer to the header of the Operating System event queue.

WaitNextEvent

You can use the `WaitNextEvent` function to retrieve events one at a time from the Event Manager.

```
pascal Boolean WaitNextEvent (EventMask eventMask,  
                             EventRecord *theEvent,  
                             UInt32 sleep,  
                             RgnHandle mouseRgn);
```

eventMask A value that indicates which kinds of events are to be returned. This parameter is interpreted as a sum of event mask constants. You specify the event mask using values defined by the `EventMask` enumeration (page 7-409). To accept all events, you can specify the `everyEvent` constant as the event mask.

If no event of any of the designated types is available, `WaitNextEvent` returns a null event. `WaitNextEvent` determines the next available event to return based on the `eventMask` parameter and the priority of the event.

Events not designated by the event mask remain in the event stream until retrieved by an application. Low-level events in the Operating System event queue are kept in the queue until they are retrieved by your application or another application or until the queue becomes full. Once the queue becomes full, the Operating System Event Manager begins discarding the oldest events in the queue.

theEvent A pointer to an event record for the next available event of the specified type or types. The `WaitNextEvent` function removes the returned event from the event stream and returns the information about the event in an event record. The event record includes the type of event received and other information. See “The Event Record” (page 7-414) for a description of the fields in the event record.

In addition to the event record, high-level events can contain additional data; you use the `AcceptHighLevelEvent` (page 7-429) function or the Apple Event Manager `AEProcessAppleEvent` function to get additional data associated with these events.

Event Manager

sleep The number of ticks (a tick is approximately $1/60$ of a second) indicating the amount of time your application is willing to relinquish the processor if no events (other than null events) are pending for your application. If you specify a value greater than 0 for the `sleep` parameter, your application relinquishes the processor for the specified time or until an event occurs.

You should usually specify a value greater than 0 for the `sleep` parameter to allow background processes to receive processing time. You should not set the `sleep` parameter to a value greater than the number of ticks returned by `GetCaretTime` (page 7-460) if your application provides text-editing capabilities. When the specified time expires, and if there are no pending events for your application, `WaitNextEvent` returns a null event in the parameter `theEvent`.

mouseRgn A handle to a region that specifies a region inside of which mouse movement does not cause mouse-moved events. In other words, your application receives mouse-moved events only when the cursor is outside the specified region. You should specify the region in global coordinates. If you pass an empty region or a `nil` region handle, the Event Manager does not report mouse-moved events to your application. Note that your application should recalculate the `mouseRgn` parameter when it receives a mouse-moved event, or it will continue to receive mouse-moved events as long as the cursor position is outside the original `mouseRgn`.

function result The `WaitNextEvent` function returns `false` as its function result if the event being returned is a null event or if `WaitNextEvent` has intercepted the event; otherwise, `WaitNextEvent` returns `true`.

DESCRIPTION

The `WaitNextEvent` function calls the Operating System Event Manager function `SystemEvent` to determine whether the event should be handled by the application or the Operating System.

If no events are pending for your application, `WaitNextEvent` waits for a specified amount of time for an event. (During this time, processing time may be allocated to background processes.) If an event occurs, it is returned through the parameter `theEvent`, and `WaitNextEvent` returns a function result of `true`. If

Event Manager

the specified time expires and there are no pending events for your application, `WaitNextEvent` returns a null event in `theEvent` and a function result of `false`.

Before returning an event to your application, `WaitNextEvent` performs other processing and may intercept the event.

The `WaitNextEvent` function intercepts Command–Shift–number key sequences and calls the corresponding 'FKEY' resource to perform the associated action. The Event Manager's processing of Command–Shift–number key sequences with numbers 3 through 9 can be disabled by setting the `ScrDmpEnable` global variable (a byte) to 0.

The `WaitNextEvent` function also makes the alarm go off if the alarm is set and the current time is the alarm time. The user sets the alarm using the Alarm Clock desk accessory.

The `WaitNextEvent` function also calls the `SystemTask` function (page 7-436), which gives time to each open desk accessory or device driver to perform any periodic action defined for it. A desk accessory or device driver specifies how often the periodic action should occur, and `SystemTask` gives time to the desk accessory or device driver at the appropriate interval.

Some high-level events may be fully specified by their event records only, while others may include additional information in an optional buffer. To get any additional information and to find the sender of the event, use the `AcceptHighLevelEvent` function (page 7-429).

If the returned event is a high-level event and your application supports Apple events, use the Apple Event Manager function `AEProcessAppleEvent` to respond to the Apple event and to get additional information associated with the Apple event.

SPECIAL CONSIDERATIONS

In System 7, if your application is in the foreground and the user opens a desk accessory or other item from the Apple menu, clicks in the window belonging to another application or desk accessory, or chooses another process from the Application menu, the next event reported to your application by the `WaitNextEvent` function is a suspend event. After your application is switched out, the Event Manager directs events (other than events your application can receive in the background) to the newly activated process until the user switches back to your application or another application.

Note

In a single-application environment in System 6, and in a multiple-application environment in which the desk accessory is launched in the application's partition (for example, a desk accessory opened by the user from the Apple menu while holding down the Option key), the Event Manager handles events for desk accessories in a slightly different manner.

In these environments, when mouse-up, activate, update, and keyboard events (including keyboard equivalents of menu commands) occur, the Event Manager checks to see whether the active window belongs to a desk accessory and whether the desk accessory can handle the event. If so, it sends the event to the desk accessory and `WaitNextEvent` returns `false` to your application. Also note that in these environments, the Event Manager returns `true` for mouse-down events, regardless of whether the mouse-down event is for a desk accessory or not. For mouse-down events in these situations, if the mouse button was pressed while the cursor was in a desk accessory window (as indicated by the `inSystem` constant returned by the Window Manager `FindWindow` function), your application should call the `SystemClick` function (page 7-435). The `SystemClick` function handles mouse-down events as appropriate for desk accessories, including sending your application an activate event to deactivate its front window if the desk accessory window needs to be activated. ♦

SEE ALSO

To get information about the sender of a high-level event and to retrieve any additional data associated with the high-level event, use the `AcceptHighLevelEvent` function (page 7-429). For details on how to process an Apple event, see the description of the `AEProcessAppleEvent` function in *Inside Macintosh: Interapplication Communication*.

To retrieve an event without removing it from the event stream, use the `EventAvail` function (page 7-427).

EventAvail

You can use the `EventAvail` function to retrieve the next available event from the Event Manager without removing the returned event from your application's event stream.

```
pascal Boolean EventAvail (EventMask eventMask,
                          EventRecord *theEvent);
```

eventMask A value that indicates which kinds of events are to be returned; this parameter is interpreted as a sum of event mask constants. You specify the event mask using one or more of the values defined by the `EventMask` enumeration (page 7-409). If no event of any of the designated types is available, `EventAvail` returns a null event.

theEvent A pointer to an event record for the next available event of the specified type or types. The `EventAvail` function does not remove the returned event from the event stream, but does return the information about the event in an event record. The event record includes the type of event received and other information.

function result `EventAvail` returns `false` as its function result if the event being returned is a null event; otherwise, `EventAvail` returns `true`.

DESCRIPTION

Like `WaitNextEvent` (page 7-423), the `EventAvail` function calls the `SystemTask` function (page 7-436) to give time to each open desk accessory or device driver to perform any periodic action defined for it. The `EventAvail` function also makes the alarm go off if the alarm is set and the current time is the alarm time. The user sets the alarm using the Alarm Clock desk accessory.

SPECIAL CONSIDERATIONS

If `EventAvail` returns a low-level event from the Operating System event queue, the event will not be accessible later if, in the meantime, the event queue becomes full and the event is discarded from it; however, this is not a common occurrence.

SEE ALSO

See “The Event Record” (page 7-414) for a description of the fields in the event record.

GetNextEvent

Although you should normally use `WaitNextEvent`, you can also use the `GetNextEvent` function to retrieve events one at a time from the Event Manager.

```
pascal Boolean GetNextEvent (EventMask eventMask,
                             EventRecord *theEvent);
```

<code>eventMask</code>	A value that indicates which kinds of events are to be returned; this parameter is interpreted as a sum of event mask constants. You specify the event mask using one or more of the values defined by the <code>EventMask</code> enumeration (page 7-409). If no event of any of the designated types is available, <code>GetNextEvent</code> returns a null event.
<code>theEvent</code>	A pointer to an event record for the next available event of the specified type or types. The <code>GetNextEvent</code> function removes the returned event from the event stream and returns the information about the event in an event record. The event record includes the type of event received and other information.

DESCRIPTION

`GetNextEvent` returns `false` as its function result if the event being returned is a null event or if `GetNextEvent` has intercepted the event; otherwise, `GetNextEvent` returns `true`. The `GetNextEvent` function calls the Operating System Manager function `SystemEvent` to determine whether the event should be handled by the application or the Operating System.

Like `WaitNextEvent` (page 7-423), the `GetNextEvent` function calls the `SystemTask` function (page 7-436) to give time to each open desk accessory or device driver to perform any periodic action defined for it. The `GetNextEvent` function also makes the alarm go off if the alarm is set and the current time is the alarm time. (The user sets the alarm using the Alarm Clock desk accessory.)

The `GetNextEvent` function also intercepts Command–Shift–number key sequences and calls the corresponding 'FKEY' resource to perform the associated action. The Event Manager's processing of Command–Shift–number key sequences with numbers 3 through 9 can be disabled by setting the `ScrDmpEnable` global variable (a byte) to 0.

SPECIAL CONSIDERATIONS

For greater support of the multitasking environment, your application should use `WaitNextEvent` instead of `GetNextEvent` whenever possible. If your application does call `GetNextEvent`, it should also call the `SystemTask` function.

SEE ALSO

See “The Event Record” (page 7-414) for a description of the fields in the event record.

AcceptHighLevelEvent

After receiving a high-level event (other than an Apple event), use the `AcceptHighLevelEvent` function to get any additional information associated with the event.

```
pascal OSErr AcceptHighLevelEvent (TargetID *sender,
                                   unsigned long *msgRefcon,
                                   void *msgBuff,
                                   unsigned long *msgLen);
```

<code>sender</code>	A pointer to a structure of type <code>TargetID</code> (page 7-417) whose contents identify the sender of the event. The structure referenced through the <code>sender</code> parameter contains the session reference number that identifies the connection with the other application and the port name and location name of the sender.
<code>msgRefcon</code>	A pointer to a value that uniquely identifies the communication associated with this event. If you send a response to this event, you should specify the same value as that referenced through the <code>msgRefcon</code> parameter so that the sender of the event can associate the reply with the original request.

Event Manager

<code>msgBuff</code>	<p>A pointer to a block of memory where the <code>AcceptHighLevelEvent</code> function should return any additional data associated with the event. Your application is responsible for allocating the memory for the additional data pointed to by the <code>msgBuff</code> parameter and for setting the <code>msgLen</code> parameter to the number of bytes that you have allocated for the data.</p> <p>If the <code>msgBuff</code> parameter points to an area in memory that is not large enough to hold all the data associated with the event, <code>AcceptHighLevelEvent</code> returns as much data as the specified memory area can hold, returns the amount of data remaining in the <code>msgLen</code> parameter, and returns the result code <code>bufferIsSmall</code>.</p>
<code>msgLen</code>	<p>A pointer to a value that specifies the size of the data (in bytes) pointed to by the <code>msgBuff</code> parameter. If <code>AcceptHighLevelEvent</code> returns the result code <code>bufferIsSmall</code>, the value referenced through the <code>msgLen</code> parameter contains the number of bytes remaining. You can call <code>AcceptHighLevelEvent</code> again to receive the rest of the data.</p>

DESCRIPTION

When your application receives a high-level event, you can use the `AcceptHighLevelEvent` function to get additional data associated with the event. The `AcceptHighLevelEvent` function returns information that identifies the sender of the event and the unique message reference constant of the event.

Your application should allocate memory for any additional data associated with the event, then supply a pointer to the data area and also provide the length in bytes of the data area.

SPECIAL CONSIDERATIONS

The `AcceptHighLevelEvent` function may move or purge memory. You should not call this function from within an interrupt, such as in a completion function or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `AcceptHighLevelEvent` function are

Trap macro	Selector
<code>_OSDispatch</code>	<code>\$0033</code>

RESULT CODES

<code>noErr</code>	0	No error
<code>bufferIsSmall</code>	-607	Buffer is too small
<code>noOutstandingHLE</code>	-608	No outstanding high-level event

SEE ALSO

For details on how to process an Apple event using the `AEProcessAppleEvent` function, see *Inside Macintosh: Interapplication Communication*.

GetSpecificHighLevelEvent

You can use the `GetSpecificHighLevelEvent` function to select and optionally retrieve a specific high-level event from your application's high-level event queue.

```
pascal Boolean GetSpecificHighLevelEvent (
    GetSpecificFilterUPP aFilter,
    void *contextPtr,
    OSErr *err);
```

aFilter A universal procedure pointer to the application-defined filter function that `GetSpecificHighLevelEvent` should use to search for a specific event; see “Filter Function Pointer and Macro” (page 7-420) for details. `GetSpecificHighLevelEvent` calls your filter function once for each event in your application's high-level event queue until the function returns `true` or the end of the queue is reached.

Event Manager

<code>contextPtr</code>	A pointer to a value that specifies the criteria your filter function should use to select a specific event. For example, you can specify the address of a reference constant to search for a particular event, the address of a target ID structure to search for a specific sender of an event, or the address of an event class to search for a specific class of event.
<code>err</code>	<code>GetSpecificHighLevelEvent</code> returns, through this parameter, a value that indicates whether any errors occurred. The <code>err</code> parameter specifies the <code>noErr</code> constant if no errors occurred or <code>noOutstandingHLE</code> if no high-level events are pending in your application's high-level event queue.

DESCRIPTION

You can use the `GetSpecificHighLevelEvent` function to search for a specific high-level event in your application's high-level event queue. You specify a filter function as one of the parameters to `GetSpecificHighLevelEvent`. The `GetSpecificHighLevelEvent` function calls your filter function once for every event in your application's high-level event queue, until your filter function returns `true` or the end of the queue is reached.

The `GetSpecificHighLevelEvent` function passes the value referenced by the `contextPtr` parameter to your filter function. Your filter function also receives as parameters the event record associated with the high-level event and the target ID structure that identifies the sender of the event. Your filter function can compare the value referenced by the `contextPtr` parameter with any of the other information it receives.

If your filter function finds a match, it can call `AcceptHighLevelEvent` (page 7-429) if necessary, and then return `true`. If your filter function does not find a match, then it should return `false`.

If your filter function returns `true`, the `GetSpecificHighLevelEvent` function returns `true`. If your filter function returns `false` for all high-level events in your application's event queue, or if there are no high-level events in the queue, `GetSpecificHighLevelEvent` returns `false`.

SPECIAL CONSIDERATIONS

The `GetSpecificHighLevelEvent` function may move or purge memory. You should not call this function from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetSpecificHighLevelEvent` function are

Trap macro	Selector
<code>_OSDispatch</code>	<code>\$0045</code>

SEE ALSO

See “Filter Function for Searching the High-Level Event Queue” (page 7-467) for more information about how to define a filter function and the parameters that `GetSpecificHighLevelEvent` passes to your filter function.

FlushEvents

The `FlushEvents` function removes low-level events from the Operating System event queue. Note that `FlushEvents` does not remove any types of events not stored in the Operating System event queue.

You can choose to use the `FlushEvents` function when your application first starts to empty the Operating System event queue of any keystrokes or mouse events generated by the user while the Finder loaded your application. In general, however, your application should not empty the queue at any other time as this loses user actions and makes your application and the computer appear unresponsive to the user.

```
pascal void FlushEvents (EventMask whichMask, EventMask stopMask);
```

<code>whichMask</code>	A value that indicates which kinds of low-level events are to be removed from the Operating System event queue; this parameter is interpreted as a sum of event mask constants. You
------------------------	---

specify the event mask using one or more of the values defined by the `EventMask` enumeration (page 7-409). The `whichMask` and `stopMask` parameters together specify which events to remove.

`stopMask`

A value that limits which low-level events are to be removed from the Operating System event queue; this parameter is interpreted as a sum of event mask constants. You specify the event mask using one or more of the values defined by the `EventMask` enumeration (page 7-409). `FlushEvents` does not remove any low-level events that are specified by the `stopMask` parameter. To remove all events specified by the `whichMask` parameter, specify 0 as the `stopMask` parameter.

DESCRIPTION

`FlushEvents` removes only low-level events stored in the Operating System event queue; it does not remove activate, update, operating-system, or high-level events.

You specify which low-level events to remove using the `whichMask` and `stopMask` parameters. `FlushEvents` removes the low-level events specified by the `whichMask` parameter, up to but not including the first event of any type specified by the `stopMask` parameter.

If the event queue doesn't contain any of the events specified by the `whichMask` parameter, `FlushEvents` does not remove any events from the queue.

ASSEMBLY-LANGUAGE INFORMATION

You must set up register D0 with the event mask (`whichMask`) and stop mask before calling `FlushEvents`. When `FlushEvents` returns, register D0 contains 0 if all events were removed from the queue or, if all events were not removed from the queue, an event code that specifies the type of event that caused the removal process to stop.

Registers on entry

D0	Event mask (low-order word)
	Stop mask (high-order word)

Registers on exit

D0 0 if all events were removed from the queue, or the event code of the event that stopped the search (low-order word)

SystemClick

After receiving a mouse-down event, your application should call the Window Manager function `FindWindow` (page 8-517) to determine where the cursor was when the mouse button was pressed. If `FindWindow` returns the `inSysWindow` constant, call the `SystemClick` function to handle the event.

```
pascal void SystemClick (const EventRecord *theEvent,
                        WindowPtr theWindow);
```

`theEvent` A pointer to the event record for the event.

`theWindow` A reference to the window in which the mouse-down event occurred. Pass the window pointer returned by `FindWindow` in this parameter.

DESCRIPTION

If a mouse-down event occurred in a desk accessory's window, the `SystemClick` function determines which part of the desk accessory's window the cursor was in when the mouse button was pressed and routes the event to the appropriate desk accessory as necessary.

If the mouse button was pressed while the cursor was in the content region of the desk accessory's window and the window is active, `SystemClick` sends the mouse-down event to the desk accessory to process. If the mouse-down event occurred in the content region of the window and the window is inactive, `SystemClick` makes it the active window. It does this by sending your application an activate event to deactivate its front window and directing an event to the desk accessory to activate its window.

If the mouse button was pressed while the cursor was in the drag region or go-away region, `SystemClick` calls the Window Manager function `DragWindow` (page 8-519) or `TrackGoAway` (page 8-531), as appropriate. If `TrackGoAway` reports

that the user closed the desk accessory, `SystemClick` sends a close message to the desk accessory.

SEE ALSO

See “The Event Record” (page 7-414) for a description of the fields in the event record.

SystemTask

In a multiple-application environment, the `WaitNextEvent` function (page 7-423) is responsible for giving time to each open desk accessory or driver to perform any periodic action. You should not call `SystemTask` if your application calls `WaitNextEvent`.

If your application calls `GetNextEvent` (page 7-428), your application should call the `SystemTask` function.

```
pascal void SystemTask (void);
```

DESCRIPTION

The `SystemTask` function gives time to each open desk accessory or driver to perform the periodic action defined for it. A desk accessory or device driver specifies how often the periodic action should occur, and `SystemTask` gives time to the desk accessory or device driver at the appropriate interval.

If your application calls `GetNextEvent` (page 7-428), your application should call `SystemTask` at least every sixtieth of a second. This usually corresponds to calling `SystemTask` once each time through your event loop. If your application does a large amount of processing, you may need to call `SystemTask` more than once in your event loop.

SystemEvent

The `WaitNextEvent` (page 7-423) and `GetNextEvent` (page 7-428) functions call the `SystemEvent` function. In most cases your application should not call the `SystemEvent` function.

The `SystemEvent` function determines if a specific event should be handled by the application or the Operating System.

```
pascal Boolean SystemEvent (const EventRecord *theEvent);
```

`theEvent` A pointer to the event record for the event.

DESCRIPTION

`SystemEvent` returns `false` as its function result if the event should be handled by the application; otherwise, `SystemEvent` takes any appropriate actions and returns `true`.

For activate, update, mouse-up, and keyboard events (including keyboard equivalents of commands), `SystemEvent` checks to see whether the active window belongs to a desk accessory and whether that desk accessory can handle that type of event. If so, `SystemEvent` sends the event to the desk accessory and returns `true`. Otherwise, `SystemEvent` returns `false`.

For mouse-down events and null events, `SystemEvent` returns `false`.

For disk-inserted events, `SystemEvent` attempts to mount the disk using the `PBMountVol` function but returns `false` so that the application can perform further processing if necessary.

ASSEMBLY-LANGUAGE INFORMATION

If the `SEvtEnb` global variable (a byte) contains 0, `SystemEvent` always returns `false`.

SEE ALSO

See “The Event Record” (page 7-414) for a description of the fields in the event record. For a description of the `PBMountVol` function, see the chapter “File Manager” in *Inside Macintosh: Files*.

GetOSEvent

The Toolbox Event Manager calls the `GetOSEvent` function to retrieve low-level events stored in the Operating System event queue. In most cases, your application should not use this function.

```
pascal Boolean GetOSEvent (EventMask mask, EventRecord *theEvent);
```

<code>mask</code>	A value that indicates which kinds of events are to be returned; this parameter is interpreted as a sum of event mask constants. You specify the event mask using one or more values defined by the <code>EventMask</code> enumeration (page 7-409). <code>GetOSEvent</code> returns only low-level events stored in the Operating System event queue; it does not return activate, update, operating-system, or high-level events. If no low-level event of any of the designated kinds is available, <code>GetOSEvent</code> returns a null event.
<code>theEvent</code>	A pointer to an event record for the next available low-level event of the specified type or types in the Operating System event queue. The <code>GetOSEvent</code> function removes the returned event from the Operating System event queue and returns the information about the event in an event record. The event record includes the type of event received and other information.

DESCRIPTION

The `GetOSEvent` function retrieves and removes an event from the Operating System event queue. `GetOSEvent` returns `false` as its function result if the event being returned is a null event; otherwise, `GetOSEvent` returns `true`. `GetOSEvent` does not intercept or respond to the event in any way. It also does not process Command-Shift- number key combinations or process any alarms set by the user through the Alarm Clock desk accessory.

ASSEMBLY-LANGUAGE INFORMATION

You must set up register A0 with the address of an event record and register D0 with the event mask before invoking `GetOSEvent`. When `GetOSEvent` returns,

register D0 indicates whether the returned event is a null event or an event other than a null event and the returned event is accessible through register A0.

Registers on entry

A0 Address of event record
D0 Event mask (low-order word)

Registers on exit

A0 Address of event record
D0 0 if `GetOSEvent` returns any event other than a null event, or -1 if it returns a null event (low-order byte)

SEE ALSO

See “The Event Record” (page 7-414) for a description of the fields in the event record.

OSEventAvail

The Toolbox Event Manager uses the `OSEventAvail` function to retrieve an event from the Operating System event queue without removing it. In most cases your application does not need to use this function.

```
pascal Boolean OSEventAvail (EventMask mask, EventRecord *theEvent);
```

`mask` A value of type `EventMask` (page 7-423) that indicates which kinds of events are to be returned; this parameter is interpreted as a sum of event mask constants. You specify the event mask using one or more values defined by the `EventMask` enumeration (page 7-409). `OSEventAvail` returns only low-level events stored in the Operating System event queue; it does not return activate, update, operating-system, or high-level events. If no low-level event of any of the designated types is available, `OSEventAvail` returns a null event.

Event Manager

`theEvent` A pointer to an event record for the next available event of the specified type or types. The `OSEventAvail` function does not remove the returned event from the Operating System event queue but does return information about the event in an event record. The event record includes the type of event received and other information.

DESCRIPTION

The `OSEventAvail` function retrieves an event from the Operating System event queue without removing it from the queue. The `OSEventAvail` function returns `false` as its function result if the event being returned is a null event; otherwise, `OSEventAvail` returns `true`.

`OSEventAvail` does not intercept or respond to the event in any way. It also does not process Command–Shift–number key combinations or process any alarms set by the user through the Alarm Clock desk accessory.

SPECIAL CONSIDERATIONS

If the `OSEventAvail` function returns a low-level event from the Operating System event queue, the event will not be accessible later if, in the meantime, the event queue becomes full and the event is discarded from it; however, this is not a common occurrence.

ASSEMBLY-LANGUAGE INFORMATION

You must set up register A0 with the address of an event record and register D0 with the event mask before invoking `OSEventAvail`. When `OSEventAvail` returns, register D0 indicates whether the returned event is a null event or some other event, and the returned event is accessible through register A0.

Registers on entry

A0	Address of event record
D0	Event mask (low-order word)

Registers on exit

- A0 Address of event record
- D0 0 if `OSEventAvail` returns any event other than a null event, or
 -1 if it returns a null event (low-order byte)

SEE ALSO

See “The Event Record” (page 7-414) for a description of the fields in the event record.

SetEventMask

The `SetEventMask` function sets the system event mask of your application to the specified mask. Your application should not call the `SetEventMask` function to disable any event types from being posted. Use `SetEventMask` only to enable key-up events if your application needs to respond to key-up events.

```
pascal void SetEventMask (EventMask value);
```

value A value that specifies which events should be posted in the Operating System event queue. You specify the event mask using one or more of the values defined by the `EventMask` enumeration (page 7-409).

DESCRIPTION

The `SetEventMask` function sets the system event mask of your application according to the parameter `value`. The Operating System Event Manager posts only low-level events (other than update or activate events) corresponding to bits in the system event mask of the current process when posting events in the Operating System event queue. The system event mask of an application is initially set to post mouse-up, mouse-down, key-down, auto-key, and disk-inserted events into the Operating System event queue.

ASSEMBLY-LANGUAGE INFORMATION

The system event mask of the current application is available in the `SysEvtMask` system global variable.

GetEvQHdr

The Event Manager uses the `GetEvQHdr` function to get a pointer to the header of the Operating System event queue. In most cases, your application should not call the `GetEvQHdr` function.

```
pascal QHdrPtr GetEvQHdr (void);
```

DESCRIPTION

The `GetEvQHdr` function returns a pointer to the header of the Operating System event queue.

ASSEMBLY-LANGUAGE NOTE

The `EventQueue` system global variable contains the header of the event queue.

SEE ALSO

See “The Event Queue” (page 7-419) for information on the structure of the Operating System event queue.

LMSetEventQueue

```
pascal void LMSetEventQueue(QHdrPtr eventQueueValue);
```

Sending Events

You can send events to other applications or processes using the `PostHighLevelEvent` function (page 7-443). To send Apple events to other applications, use the Apple Event Manager function `AESend`. The Operating

System Event Manager also provides the `PPostEvent` (page 7-446) and `PostEvent` (page 7-448) functions for posting low-level events to the Operating System event queue. The `PostEvent` function is used by the Toolbox Event Manager. In most cases, your application should not use the `PostEvent` function.

PostHighLevelEvent

You can use the `PostHighLevelEvent` function to send a high-level event to another application.

```
pascal OSErr PostHighLevelEvent (EventRecord *theEvent,
                                unsigned long receiverID,
                                unsigned long msgRefcon,
                                void *msgBuff,
                                unsigned long msgLen,
                                unsigned long postingOptions);
```

`theEvent`

A pointer to the event record for the event to send. Your application should fill out the `what`, `message`, and `where` fields of the event record. Specify the `kHighLevelEvent` constant in the `what` field, the event class of the high-level event in the `message` field, and the event ID in the `where` field. You do not need to fill out the `when` or `modifiers` fields; the Event Manager automatically assigns the appropriate values to these fields when you send the message.

`receiverID`

The recipient of the high-level event. When sending an event to another application on the local computer, you can specify the recipient of the event by session reference number, process serial number, signature, or port name and location name. When sending an event to an application on a remote computer, you can specify the recipient only by the session reference number or by the port name and location name.

To specify a port name and location name, provide the address of a target ID structure in the `receiverID` parameter. To specify a process serial number, provide its address in the `receiverID` parameter. To specify a session reference number, or signature, provide the data in the `receiverID` parameter.

Event Manager

<code>msgRefcon</code>	A unique number that identifies the communication associated with this event. Your application can set this field to any value it chooses. If you are replying to a high-level event, you should use the same value in the <code>msgRefcon</code> parameter as specified in the high-level event that originated the request.
<code>msgBuff</code>	A pointer to a data buffer that contains any additional data for the event.
<code>msgLen</code>	The size (in bytes) of the data buffer pointed to by the <code>msgBuff</code> parameter.
<code>postingOptions</code>	Options associated with the <code>receiverID</code> parameter and delivery options associated with the event. You can specify one or more delivery options to indicate whether you want the other application to receive the event at the next opportunity and to indicate whether you want acknowledgment that the event was received by the other application. You use the options associated with the <code>receiverID</code> parameter to indicate how you are specifying the recipient of the event—whether by port name and location name in a target ID structure, by session reference number, by process serial number, or by signature. For descriptions of the enumerators you can use to specify posting options, see “Posting Options” (page 7-412).

DESCRIPTION

The `PostHighLevelEvent` function posts the high-level event to the specified process.

If the application to which you are sending a high-level event terminates, you receive the result code `sessionClosedErr` the next time your application calls `PostHighLevelEvent` to send another high-level event to the terminated application. If you do not care about any state information about that session, you can just resend your event. Otherwise, you must restart another session and resend your event.

If your application is running in the background and posts a high-level event that requires the network authentication dialog box to be displayed, `PostHighLevelEvent` returns the `noUserInteractionAllowed` result code, does not display the network authentication dialog box, and does not send the event. If your application receives the `noUserInteractionAllowed` result code, you can use

the Notification Manager to inform the user that your application needs attention. When the user brings your application to the foreground, you can repost the event. If the reposting is successful, your application can continue to post high-level events without further user interaction. Note that `PostHighLevelEvent` can return `noUserInteractionAllowed` only on the first posting of a high-level event to a remote target.

SPECIAL CONSIDERATIONS

The `PostHighLevelEvent` function may move or purge memory. You should not call this function from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PostHighLevelEvent` function are

Trap macro	Selector
<code>_OSDispatch</code>	<code>\$0034</code>

SEE ALSO

For details on how to send Apple events to other applications using the `AEsend` function, see *Inside Macintosh: Interapplication Communication*.

RESULT CODES

noErr	0	No error
connectionInvalid	-609	Connection is invalid
noUserInteractionAllowed	-610	Cannot interact directly with user
sessionClosedErr	-917	Session closed

PPostEvent

In most cases, your application does not need to post events in the Operating System event queue; however, if you must do so, you can use the `PPostEvent` function.

```
pascal OSErr PPostEvent (EventKind eventCode,  
                        UInt32 eventMsg,  
                        EvQElPtr *qEl);
```

eventCode	A value of type <code>EventKind</code> (page 7-414) that indicates the type of event to post into the Operating System event queue. You specify the event kind using one or more of these values defined by the <code>EventMask</code> enumeration (page 7-409): <code>mouseDown</code> , <code>mouseUp</code> , <code>keyDown</code> , <code>keyUp</code> , <code>autoKey</code> , and <code>diskEvt</code> . Do not attempt to post any other type of event in the Operating System event queue.
eventMsg	An unsigned 32-bit integer that contains the contents of the <code>message</code> field for the event that <code>PPostEvent</code> should post in the queue.
qEl	You specify the address of a pointer to an event queue entry in this parameter. <code>PPostEvent</code> returns the event queue entry of the posted event through this parameter.

DESCRIPTION

In the `eventCode` and `eventMsg` parameters, you specify the value for the `what` and `message` fields of the event's event record. The `PPostEvent` function fills out the `when`, `where`, and `modifiers` fields of the event record with the current time, current mouse location, and current state of the modifier keys and mouse button.

Event Manager

The `PPostEvent` function returns, through the `qEl` parameter, a pointer to the event queue entry of the posted event. You can change any fields of the posted event by changing the fields of its event queue entry. For example, you can change the posted event's modifier keys by changing the value of the `evtQModifiers` field of the event queue entry.

The `PPostEvent` function posts only events that are enabled by the system event mask. If the event queue is full, `PPostEvent` removes the oldest event in the queue and posts the new event.

▲ **WARNING**

Do not post any events other than mouse-down, mouse-up, key-down, key-up, auto-key, and disk-inserted events in the Operating System event queue. Attempting to post other events into the Operating System event queue interferes with the internal operation of the Event Manager.

**ASSEMBLY-LANGUAGE INFORMATION**

You must set up register A0 and register D0 before invoking `PPostEvent`. The `PPostEvent` function returns values in registers A0 and D0.

Registers on entry

A0 Event number (low-order word)
D0 Event message (long)

Registers on exit

A0 Pointer to an event queue entry (long)
D0 Result code (low-order word)

RESULT CODES

evtNotEnb	1	Event type not valid—event not posted
noErr	0	No error

SEE ALSO

For a description of the entries in the event queue, see “The Event Queue” (page 7-419).

PostEvent

The Toolbox Event Manager uses the `PostEvent` function to post events into the Operating System event queue. In most cases, your application should not call the `PostEvent` function.

```
pascal OSErr PostEvent (EventKind eventNum,  
                        UInt32 eventMsg);
```

eventNum	A value that indicates the type of event to post into the Operating System event queue. You specify the event kind using one or more of these values defined by the <code>EventMask</code> enumeration (page 7-409): <code>mouseDown</code> , <code>mouseUp</code> , <code>keyDown</code> , <code>keyUp</code> , <code>autoKey</code> , and <code>diskEvt</code> . Do not attempt to post any other type of event in the Operating System event queue.
eventMsg	An unsigned integer that contains the contents of the <code>message</code> field for the event that <code>PostEvent</code> should post in the queue.

DESCRIPTION

In the `eventNum` and `eventMsg` parameters, you specify the value for the `what` and `message` fields of the event's event record. The `PostEvent` function fills out the `when`, `where`, and `modifiers` fields of the event record with the current time, current mouse location, and current state of the modifier keys and mouse button.

The `PostEvent` function posts only events that are enabled by the system event mask. If the event queue is full, `PostEvent` removes the oldest event in the queue and posts the new event.

Event Manager

Note that if you use `PostEvent` to repost an event, the `PostEvent` function fills out the `when`, `where`, and `modifier` fields of the event record, giving these fields of the reposted event different values from the values contained in the original event.

▲ **WARNING**

Do not post any events other than mouse-down, mouse-up, key-down, key-up, auto-key, and disk-inserted events in the Operating System event queue. Attempting to post other events into the Operating System event queue interferes with the internal operation of the Event Manager.

**ASSEMBLY-LANGUAGE INFORMATION**

You must set up register A0 with the event code and register D0 with the event message before invoking `PostEvent`. When `PostEvent` returns, register D0 contains the result code.

Registers on entry

A0 Event number (low-order word)

D0 Event message (long)

Registers on exit

D0 Result code (low-order word)

RESULT CODES

<code>evtNotEnb</code>	1	Event type not valid—event not posted
<code>noErr</code>	0	No error

Converting Process Serial Numbers and Port Names

The Event Manager provides two functions to convert between process serial numbers and port names: `GetProcessSerialNumberFromPortName` (page 7-450) and `GetPortNameFromProcessSerialNumber` (page 7-451). Both functions are intended to map serial numbers to port names (or vice versa) for applications

open on the local computer. They do not return useful results for applications open on remote computers.

GetProcessSerialNumberFromPortName

Use `GetProcessSerialNumberFromPortName` to get the process serial number of a process.

```
pascal OSErr GetProcessSerialNumberFromPortName
              (const PPCPortRec *portName,
               ProcessSerialNumber *pPSN);
```

portName A pointer to a PPC port record, the contents of which specify the port name registered to a process whose serial number you want.

pPSN Returns a pointer to the process serial number of the process designated through the `portName` parameter. You can use the returned process serial number to send a high-level event to that process. Do not interpret the value of the process serial number.

DESCRIPTION

The `GetProcessSerialNumberFromPortName` function returns, in the `pPSN` parameter, a pointer to the process serial number of the process registered at a specific port.

SPECIAL CONSIDERATIONS

The `GetProcessSerialNumberFromPortName` function does not move or purge memory but for other reasons should not be called from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the
GetProcessSerialNumberFromPortName function are

Trap macro	Selector
_OSDispatch	\$0035

RESULT CODES

noErr	0	No error
noPortErr	-903	Invalid port name

SEE ALSO

For a description of the PPCPortRec data type, see the chapter
“Program-to-Program Communications Toolbox” in *Inside Macintosh:
Interapplication Communication*.

GetPortNameFromProcessSerialNumber

Use GetPortNameFromProcessSerialNumber to get the port name of a process.

```
pascal OSErr GetPortNameFromProcessSerialNumber
    (PPCPortRec *portName,
     const ProcessSerialNumber *pPSN);
```

portName Returns a pointer to a PPC port record, the contents of which specify the port name of the process designated by the pPSN parameter. You can use the returned port name to send a high-level event to that process.

PSN A pointer to the process serial number of the process whose port name you want.

DESCRIPTION

The `GetPortNameFromProcessSerialNumber` function returns, through the `portName` parameter, the port name registered to a process having a specific process serial number.

SPECIAL CONSIDERATIONS

The `GetPortNameFromProcessSerialNumber` function does not move or purge memory but for other reasons should not be called from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetPortNameFromProcessSerialNumber` function are

Trap macro	Selector
<code>_OSDispatch</code>	<code>\$0046</code>

RESULT CODES

<code>noErr</code>	0	No error
<code>procNotFound</code>	-600	No eligible process with specified process serial number

SEE ALSO

For a description of the `PPCPortRec` data type, see the chapter “Program-to-Program Communications Toolbox” in *Inside Macintosh: Interapplication Communication*.

Reading the Mouse

The Event Manager provides functions you can use to get information about the mouse. You can get the current mouse location using the `GetMouse` function (page 7-453). You can use the `Button` function (page 7-453) to determine whether the user pressed the mouse button. After receiving a mouse-down event, you can use the `StillDown` function (page 7-454) to determine whether the mouse

button is still down, and you can use `WaitMouseUp` (page 7-454) to determine if the user subsequently released the mouse.

GetMouse

You can use the `GetMouse` function to obtain the current mouse location.

```
pascal void GetMouse (Point *mouseLoc);
```

`mouseLoc` Returns a pointer to a point describing the current mouse location in local coordinates of the current graphics port (for example, the active window). Note that this value differs from the value of the `where` field of the event record, which specifies the mouse location in global coordinates.

Button

You can use the `Button` function to determine whether the user pressed the mouse button.

```
pascal Boolean Button (void);
```

DESCRIPTION

The `Button` function looks in the Operating System event queue for a mouse-down event. If it finds one, the `Button` function returns `true`; otherwise, it returns `false`. To determine whether the mouse button is still down after a mouse-down event, use the `StillDown` function.

SEE ALSO

See “The Event Queue” (page 7-419) for information about the Operating System event queue.

StillDown

After receiving a mouse-down event, you can use the `StillDown` function to determine if the mouse button is still down.

```
pascal Boolean StillDown (void);
```

DESCRIPTION

The `StillDown` function looks in the Operating System event queue for a mouse event. If it finds one, the `StillDown` function returns `false`. If it does not find any mouse events pending in the Operating System event queue, the `StillDown` function returns `true`.

SEE ALSO

See “The Event Queue” (page 7-419) for information about the Operating System event queue.

WaitMouseUp

After receiving a mouse-down event, you can use the `WaitMouseUp` function to determine if the user subsequently released the mouse.

```
pascal Boolean WaitMouseUp (void);
```

DESCRIPTION

The `WaitMouseUp` function looks in the Operating System event queue for a mouse-up event. If it finds one, the `WaitMouseUp` function removes the mouse-up event from the queue and returns `false`. If it does not find any mouse up events pending in the Operating System event queue, the `WaitMouseUp` function returns `true`.

SEE ALSO

See “The Event Queue” (page 7-419) for information about the Operating System event queue.

Reading the Keyboard

The Event Manager reports keyboard events one at a time at your application’s request when you use the `WaitNextEvent` (page 7-423), `EventAvail` (page 7-427), or `GetNextEvent` (page 7-428) function. In addition to getting keyboard events when the user presses or releases a key, you can directly read the keyboard (and keypad) at any time using the `GetKeys` function (page 7-455).

You can also use the `KeyTranslate` function (page 7-456) to convert virtual key codes to character code values using a specified 'KCHR' resource.

GetKeys

You can use the `GetKeys` function to obtain the current state of the keyboard.

```
pascal void GetKeys (KeyMap theKeys);
```

theKeys Returns the current state of the keyboard, including the keypad, if any. The `GetKeys` function returns this information using the `KeyMap` type.

```
typedef long KeyMap[4];
```

The `KeyMap` type is interpreted as an array of 128 elements, each having a Boolean value. Each key on the keyboard or keypad corresponds to an element in the `KeyMap` array. The index for a particular key is the same as the key’s virtual key code minus 1. For example, the key with virtual key code 38 (the “J” key on the Apple Keyboard II) is the 38th element in the returned array. A `KeyMap` element is `true` if the corresponding key is down and `false` if it isn’t. The maximum number of keys that can be down simultaneously is two character keys plus any combination of the five modifier keys.

DESCRIPTION

You can use the `GetKeys` function to determine the current state of the keyboard at any time. For example, you can determine whether one of the modifier keys is down by itself or in combination with another key using the `GetKeys` function.

KeyTranslate

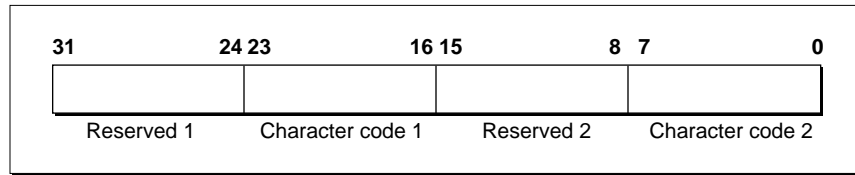
You can use the `KeyTranslate` function to convert a virtual key code to a character code based on a 'KCHR' resource. The `KeyTranslate` function is also available as the `KeyTrans` function.

```
pascal long KeyTranslate (const void *transData,
                        UInt16 keycode,
                        UInt32 *state);
```

<code>transData</code>	A pointer to the 'KCHR' resource that you want the <code>KeyTranslate</code> function to use when converting the key code to a character code.
<code>keycode</code>	A 16-bit value that your application should set so that bits 0–6 contain the virtual key code and bit 7 contains either 1 to indicate an up stroke or 0 to indicate a down stroke of the key. Bits 8–15 have the same interpretation as the high byte of the <code>modifiers</code> field of the event record and should be set according to the needs of your application.
<code>state</code>	A pointer to a value that your application should set to 0 the first time it calls <code>KeyTranslate</code> or any time your application calls <code>KeyTranslate</code> with a different 'KCHR' resource. Thereafter, your application should pass the same value in the <code>state</code> parameter as <code>KeyTranslate</code> returned in the previous call.

DESCRIPTION

The `KeyTranslate` function returns a 32-bit value that gives the character code for the virtual key code specified by the `keycode` parameter. Figure 7-1 shows the structure of the 32-bit number that `KeyTranslate` returns.

Figure 7-1 Structure of the `KeyTranslate` function result

The `KeyTranslate` function returns the values that correspond to one or possibly two characters that are generated by the specified virtual key code. For example, a given virtual key code might correspond to an alphabetic character with a separate accent character. For example, when the user presses Option-E followed by N, you can map this through the `KeyTranslate` function using the U.S. 'KCHR' resource to produce 'n, which `KeyTranslate` returns as two characters in the bytes labeled Character code 1 and Character code 2. If `KeyTranslate` returns only one character code, it is always in the byte labeled Character code 2. However, your application should always check both bytes labeled Character code 1 and Character code 2 in Figure 7-1 for possible values that map to the virtual key code.

SEE ALSO

For additional information on the 'KCHR' resource and the `KeyTranslate` function, see *Inside Macintosh: Text*.

Getting Timing Information

You can get the current number of ticks since the system last started up using the `TickCount` function (page 7-458). You can use this function to compare the number of ticks that have expired since a given event or other action occurred.

By using the `GetDblTime` function (page 7-458), you can get the suggested maximum difference in ticks that should exist to consider two mouse events a double click. The user can adjust this value using the Mouse control panel. Using the `GetCaretTime` function (page 7-460), you can get the suggested maximum difference in ticks that should exist between blinks of the caret in editable text. The user can adjust this value using the General Controls panel.

TickCount

You can use the `TickCount` function to get the current number of ticks (a tick is approximately $1/60$ of a second) since the system last started up.

```
pascal UInt32 TickCount (void);
```

DESCRIPTION

The `TickCount` function returns an unsigned 32-bit integer that indicates the current number of ticks since the system last started up. You can use this value to compare the number of ticks that have elapsed since a given event or other action occurred. For example, you could compare the current value returned by `TickCount` with the value of the `when` field of an event record.

The tick count is incremented during the vertical retrace interrupt, but this interrupt can be disabled. Your application should not rely on the tick count to increment with absolute precision. Your application also should not assume that the tick count always increments by 1; an interrupt task might keep control for more than one tick. If your application keeps track of the previous tick count and then compares this value with the current tick count, your application should compare the two values by checking for a “greater than or equal” condition rather than “equal to previous tick count plus 1.”

▲ WARNING

Don't rely on the tick count being exact; it's usually accurate to within one tick, but this level of accuracy is not guaranteed. ▲

ASSEMBLY-LANGUAGE NOTE

The value returned by `TickCount` is also accessible in the global variable `Ticks`.

GetDbtTime

To determine whether a sequence of mouse events constitutes a double click, your application measures the elapsed time (in ticks) between a mouse-up event and a mouse-down event. If the time between the two mouse events is

less than the value returned by `GetDb1Time`, your application should interpret the two mouse events as a double click.

```
pascal UInt32 GetDb1Time (void)
```

DESCRIPTION

The `GetDb1Time` function returns the suggested maximum elapsed time, in ticks, between a mouse-up event and a mouse-down event. The user can adjust this value using the Mouse control panel.

If your application distinguishes a double click of the mouse from a single click, your application should use the value returned by `GetDb1Time` to make this distinction. If your application uses `TextEdit`, the `TextEdit` functions automatically recognize and handle double clicks of text within a `TextEdit` edit record by appropriately highlighting or unhighlighting the selection.

Note

The ratio of ticks to value in the `DoubleTime` global variable is 1:1. However, the Finder multiplies `DoubleTime` by 2 to determine double click time because it needs to account for user problems that typically occur during icon arrangement. Therefore, the Finder uses `DoubleTime*2` whereas the rest of the system uses `DoubleTime`.

Incidentally, the Finder does not limit the `DoubleTime` to 64 ticks. In most places, it treats it like a byte although in some others it treats it like a longword. The best method would be to provide a one-second double-byte (two seconds in the Finder). ♦

ASSEMBLY-LANGUAGE NOTE

The value returned by `GetDb1Time` is also accessible in the system global variable `DoubleTime`.

LMSetDoubleTime

Sets the elapsed time (in ticks) between a mouse-up event and a mouse-down event.

```
pascal void LMSetDoubleTime (UInt32 value);
```

value An unsigned 32-bit integer that specifies the suggested maximum elapsed time, in ticks, between a mouse-up event and a mouse-down event. The user can adjust this value using the Mouse control panel.

DESCRIPTION

The `LMSetDb1Time` function sets the suggested maximum elapsed time, in ticks, between a mouse-up event and a mouse-down event. The user can adjust this value using the Mouse control panel.

If your application distinguishes a double click of the mouse from a single click, your application should use the value set by `LMSetDb1Time` to make this distinction. If your application uses `TextEdit`, the `TextEdit` functions automatically recognize and handle double clicks of text within a `TextEdit` edit record by appropriately highlighting or unhighlighting the selection.

ASSEMBLY-LANGUAGE NOTE

The value set by `LMSetDb1Time` is also accessible in the system global variable `DoubleTime`.

GetCaretTime

You can use the `GetCaretTime` function to get the suggested difference in ticks that should exist between blinks of the caret (usually a vertical bar marking the insertion point) in editable text. The user can adjust this value using the General Controls panel.

```
pascal UInt32 GetCaretTime (void)
```

DESCRIPTION

If your application supports editable text, your application should use the value returned by `GetCaretTime` to determine how often to blink the caret. If your application uses only `TextEdit`, you can use `TextEdit` functions to automatically blink the caret at the time interval that the user specifies in the General Controls panel.

ASSEMBLY-LANGUAGE NOTE

The value returned by `GetCaretTime` is also accessible in the system global variable `CaretTime`.

LMSetCaretTime

Sets the suggested difference in ticks that should exist between blinks of the caret (usually a vertical bar marking the insertion point) in editable text. The user can adjust this value using the General Controls panel.

```
pascal void LMSetCaretTime (UInt32 value);
```

value	An unsigned 32-bit value that specifies the difference in ticks that should exist between blinks of the caret
-------	---

DESCRIPTION

If your application supports editable text, your application should use the value returned by `LMSetCaretTime` to determine how often to blink the caret.

ASSEMBLY-LANGUAGE NOTE

The value set by `LMSetCaretTime` is also accessible in the system global variable `CaretTime`.

Accessors for Low-Memory Globals

You should never access low-memory globals directly. Instead, use a high-level Event Manager function, if one is available. If a high-level function is not available, you can use one of the accessors described here.

LMGetKeyRepThresh

Returns a value that specifies the value of the auto-key rate.

```
pascal SInt16 LMGetKeyRepThresh (void);
```

DESCRIPTION

`LMGetKeyRepThresh` returns a signed 16-bit integer that describes the value of the auto-key rate, that is, the amount of time, in ticks, that must elapse before the Event Manager generates a subsequent auto-key event.

ASSEMBLY-LANGUAGE NOTE

The value obtained by `LMGetKeyRepThresh` is also accessible in the system global variable `KeyRepThresh`.

LMSetKeyRepThresh

Sets the low-memory auto-key rate.

```
pascal void LMSetKeyRepThresh (SInt16 value);
```

<code>value</code>	A signed 16-bit integer that specifies the low-memory auto-key rate.
--------------------	--

DESCRIPTION

`LMSetKeyRepThresh` specifies a signed 16-bit integer that sets the low-memory value of the auto-key rate, that is, the amount of time, in ticks, that must elapse before the Event Manager generates a subsequent auto-key event.

ASSEMBLY-LANGUAGE NOTE

The value set by `LMSetKeyRepThresh` is also accessible in the system global variable `KeyRepThresh`.

LMGetKeyThresh

Returns a value that specifies the low-memory auto-key key threshold.

```
pascal SInt16 LMGetKeyThresh (void);
```

DESCRIPTION

`LMGetKeyThresh` returns a signed 16-bit integer that describes the value of the auto-key threshold, that is, the amount of time, in ticks, that must elapse before the Event Manager generates an auto-key event.

ASSEMBLY-LANGUAGE NOTE

The value obtained by `LMGetThresh` is also accessible in the system global variable `KeyThresh`.

LMSetKeyThresh

Sets the auto-key threshold.

```
pascal void LMSetKeyThresh (SInt16 value);
```

<code>value</code>	A signed 16-bit integer that specifies the low-memory auto-key threshold.
--------------------	---

DESCRIPTION

`LMSetKeyThresh` specifies a signed 16-bit integer that sets the low-memory auto-key threshold, that is, the amount of time, in ticks, that must elapse before the Event Manager generates an auto-key event.

ASSEMBLY-LANGUAGE NOTE

The value set by `LMSetKeyThresh` is also accessible in the system global variable `KeyRepThresh`.

LMGetSEvtEnb

Returns a value that specifies the system event enabled bit

```
pascal UInt8 LMGetSEvtEnb (void);
```

DESCRIPTION

`LMGetSEvtEnb` returns a signed 16-bit integer that describes the low-memory system event enabled bit, a byte that, if set to 0, causes the `SystemEvent` function (page 7-437) to always return `false`.

ASSEMBLY-LANGUAGE NOTE

The value obtained by `LMGetSEvtEnb` is also accessible in the system global variable `SEvtEnb`.

LMSetSEvtEnb

Sets the low-memory system event enabled bit.

```
pascal void LMSetSEvtEnb (UInt8 value);
```

<code>value</code>	An unsigned 8-bit integer that describes the value of the system event enabled bit.
--------------------	---

DESCRIPTION

`LMSetSEvtEnb` specifies an unsigned 8-bit integer that sets the low-memory system event enabled bit, , a byte that, if set to 0, causes the `SystemEvent` function (page 7-437) to always return `false`.

ASSEMBLY-LANGUAGE NOTE

The value set by `LMSetSEvtEnb` is also accessible in the system global variable `SEvtEnb`.

LMGetSysEvtMask

Returns a value that describes the low-memory system event mask.

```
pascal SInt16 LMGetSysEvtMask (void);
```

DESCRIPTION

`LMGetSysEvtMask` returns a signed 16-bit integer that describes the low-memory system event mask of the current application.

ASSEMBLY-LANGUAGE NOTE

The value obtained by `LMGetSysEvtMask` is also accessible in the system global variable `SysEvtMask`.

LMSetSysEvtMask

Sets a specified value for the system event mask.

```
pascal void LMSetSysEvtMask (SInt16 value);
```

<code>value</code>	A signed 16-bit integer that specifies the value for the system event mask of the current application.
--------------------	--

ASSEMBLY-LANGUAGE NOTE

The value specified by `LMSetSysEvtMask` is also accessible in the system global variable `SysEvtMask`.

LMGetTicks

Returns a value that describes the low-memory ticks global variable.

```
pascal SInt32 LMGetTicks (void);
```

DESCRIPTION

`LMGetTicks` returns a signed 16-bit integer that describes the current number of ticks since the system last started up. For details on functions that let you obtain data about timing, see “Getting Timing Information” (page 7-457).

ASSEMBLY-LANGUAGE NOTE

The value obtained by `LMGetTicks` is also accessible in the system global variable `Ticks`.

LMSetTicks

Specifies a value for the low-memory ticks global variable.

```
pascal void LMSetTicks (SInt32 value);
```

value	A signed 32-bit integer that specifies the number of ticks since the system last started up.
-------	--

ASSEMBLY-LANGUAGE NOTE

The value specified by `LMSetSysEvtMask` is also accessible in the system global variable `SysEvtMask`.

Application-Defined Function

When you use `GetSpecificHighLevelEvent` (page 7-431), you supply a filter function so that your application can search for a specific event in the high-level event queue of your application.

Filter Function for Searching the High-Level Event Queue

This section describes the filter function that you can provide to `GetSpecificHighLevelEvent`. For example, you might use a filter function to search for a high-level event sent from a specific application.

MyFilter

When you use `GetSpecificHighLevelEvent` (page 7-431) to search the high-level event queue of your application for a specific event, you supply a pointer to a filter function. `GetSpecificHighLevelEvent` calls your filter function once for each event in the high-level event queue until your filter function returns `true` or the end of the queue is reached. Your filter function can examine each event and determine whether that event is the desired event. If so, your filter function should return `true`.

Here's how you declare the filter function `MyFilter`:

```
pascal Boolean MyFilter (void *contextPtr,
                        HighLevelEventMsgPtr msgBuff,
                        const TargetID *sender);
```

`contextPtr` The address of data that specifies the criteria your filter function should use to select a specific event. For example, you can specify in the `contextPtr` parameter the address of a reference constant to search for a particular event, the address of a target ID structure to search for a specific sender of an event, or the address of an event class to search for a specific class of event.

Event Manager

<code>msgBuff</code>	A pointer to a structure of type <code>HighLevelEventMsg</code> , which provides: the event record for the high-level event and the reference constant of the event. The <code>HighLevelEventMsg</code> data type is described in “The High-Level Event Message Structure” (page 7-418).
<code>sender</code>	The address of the target ID structure of the application that sent the event. The target ID structure is described in “The Target ID Structure” (page 7-417).

DESCRIPTION

Your filter function can compare the contents of the `contextPtr` parameter with the contents of the `msgBuff` and `sender` parameters. If your filter function finds a match, it can call `AcceptHighLevelEvent`, if necessary, and your filter function should return `true`. If your filter function does not find a match, it should return `false`.

SEE ALSO

For information about getting a routine descriptor for your filter function, see “Filter Function Pointer and Macro” (page 7-420).

Event Manager Resource

This section explains the structure of a 'SIZE' resource and the meaning of each of its fields. You are responsible for creating the information in this resource.

The Size Resource

Every application executing in System 7, as well as every application executing under MultiFinder, should contain a size ('SIZE') resource. One of the principal functions of the 'SIZE' resource is to inform the Operating System about the memory size requirements for the application so that the Operating System can set up an appropriately sized partition for the application. The 'SIZE' resource is also used to indicate certain scheduling options to the Operating System, such as whether the application can accept suspend and resume events. The 'SIZE' resource in System 7 contains additional information indicating whether the application is 32-bit clean, whether it supports stationery documents,

whether it uses TextEdit's inline input services, whether the application wishes to receive notification of the termination of any applications it has launched, and whether the application wishes to receive high-level events.

A 'SIZE' resource consists of a 16-bit flags field followed by two 32-bit size fields. The flags field specifies operating characteristics of the application, and the size fields indicate the minimum and preferred partition sizes for the application. The minimum partition size is the actual limit below which your application will not run. The preferred partition size is the memory size at which your application can run most effectively and which the Operating System attempts to secure upon launching the application. If that amount of memory is unavailable, the application is placed into the largest contiguous block available, provided that it is larger than the specified minimum size.

Note

If the amount of available memory is between the minimum and the preferred sizes, the Finder displays a dialog box asking if the user wants to run the application using the amount of memory available. If your application does not have a 'SIZE' resource, it is assigned a default partition size of 512 KB and the Process Manager uses a default value of `false` for all specifications normally defined by constants in the flags field. ♦

When you define a 'SIZE' resource, you should give it a resource ID of -1. A user can modify the preferred size in the Finder's information window for your application. If the user does alter the preferred partition size, the Operating System creates a new 'SIZE' resource having resource ID 0. The Process Manager also creates a new 'SIZE' resource when the user modifies any of the other settings in the resource.

In system software version 7.1 the user can also modify the minimum size in the Finder's information window for your application. In version 7.1, if the user alters either the minimum or the preferred partition size, the Operating System creates two new 'SIZE' resources, one with resource ID 0 and one with resource ID 1.

At application launch time, the Process Manager looks for a 'SIZE' resource with ID 0 for the preferred partition size; if this resource is not found, it uses your original 'SIZE' resource with ID -1. In version 7.1, the Process Manager looks for a 'SIZE' resource with ID 0 for the preferred size and looks for a 'SIZE' resource with ID 1 for the minimum size; if these resources are not found, it uses your original 'SIZE' resource with ID -1.

Listing 7-1 shows the structure of the 'SIZE' resource in Rez format.

Listing 7-1 A Rez template for a 'SIZE' resource

```

type 'SIZE' {
    boolean    reserved;                /* reserved */
    boolean    ignoreSuspendResumeEvents, /* ignores suspend-resume events */
               acceptSuspendResumeEvents; /* accepts suspend-resume events */
    boolean    reserved;                /* reserved */
    boolean    cannotBackground,         /* can't use background null events */
               canBackground;            /* can use background null events */
    boolean    needsActivateOnFGSwitch,   /* needs activate event following
                                          /* major switch */
               doesActivateOnFGSwitch;    /* activates own windows in
                                          /* response to OS events */
    boolean    backgroundAndForeground,   /* app has a user interface */
               onlyBackground;            /* app has no user interface */
    boolean    dontGetFrontClicks,        /* don't return mouse events */
               getFrontClicks;            /* do return mouse events
                                          /* in front window on resume */
    boolean    ignoreAppDiedEvents,        /* applications use this */
               acceptAppDiedEvents;       /* app launchers use this */
    boolean    not32BitCompatible,         /* works with 24-bit addr */
               is32BitCompatible;         /* works with 24- or 32-bit addr */
    boolean    notHighLevelEventAware,     /* can't use high-level events */
               isHighLevelEventAware;     /* can use high-level events */
    boolean    onlyLocalHLEvents,          /* only local high-level events */
               localAndRemoteHLEvents;    /* also remote high-level events */
    boolean    notStationeryAware,         /* can't use stationery documents */
               isStationeryAware;         /* can use stationery documents */
    boolean    dontUseTextEditServices,    /* can't use inline services */
               useTextEditServices;       /* can use inline services */
    boolean    reserved;                  /* reserved */
    boolean    reserved;                  /* reserved */
    boolean    reserved;                  /* reserved */
                                          /* memory sizes are in bytes */
    unsigned longint;                     /* preferred memory size */
    unsigned longint;                     /* minimum memory size */
};

```

The nonreserved bits in the flags field have the following meanings:

Flag descriptions

`acceptSuspendResumeEvents`

When set, indicates that your application can process suspend and resume events (which the Operating System sends to your application before sending it into the background or when bringing it into the foreground).

Note

If you set the `acceptSuspendResumeEvents` flag, you should also set the `doesActivateOnFGSwitch` flag. ♦

`canBackground`

When set, indicates that your application wants to receive null event processing time while in the background. If your application has nothing to do in the background, you should not set this flag.

Note

System 7-savvy applications should have the background processing bit set. ♦

`doesActivateOnFGSwitch`

When set, indicates that your application takes responsibility for activating and deactivating any windows in response to a suspend or resume event. If the `acceptSuspendResumeEvents` flag is set, if the `doesActivateOnFGSwitch` flag is not set, and if your application is suspended, then your application receives an activate event following the suspend event. However, if you set the `doesActivateOnFGSwitch` flag, then your application won't receive activate events associated with operating-system events, and you must take care of activation and deactivation when it receives the corresponding suspend or resume event. This means that if a window of your application is frontmost, you should treat a suspend event as though a deactivate event were received as well (assuming that both the `doesActivateOnFGSwitch` and `acceptSuspendResumeEvents` flags are set). For example, you should hide scroll bars, hide any caret, and unhighlight any selected text if your application moves to the background. If you do not set this

flag, the Process Manager creates an offscreen window to force the activate and deactivate events to occur.

`onlyBackground`

When set, indicates that your application runs only in the background. Usually this is because it does not have a user interface and cannot run in the foreground.

`getFrontClicks`

When set, indicates that your application is to receive the mouse-down and mouse-up events that are used to bring your application into the foreground when the user clicks in your application's frontmost window. Typically, the user simply wants to bring your application into the foreground, so it is usually not desirable to receive the mouse events (which would probably move the insertion point or start drawing immediately, depending on the application). The Finder is one application, however, that has the `getFrontClicks` flag set.

When the user clicks in the front window of your application and causes a foreground switch, your application receives a resume event. Your application should activate its front window in response to the resume event. In this case if your application's `getFrontClicks` flag is not set, your application does not receive the associated mouse event that caused the foreground switch. If your application's `getFrontClicks` flag is set, your application does receive the associated mouse event.

Your application always receives the associated mouse event when the user clicks in one of your application's windows other than the front window and causes a foreground switch.

When your application receives a mouse-down event in System 7, your application can examine bit 0 of the `modifiers` field of the event record to determine if the mouse-down event caused a foreground switch. This information can be especially useful if your application sets its `getFrontClicks` flag. For example, your application can examine bit 0 to determine whether to process the mouse-down event (probably depending on whether the clicked item was visible before the foreground switch).

`acceptAppDiedEvents`

When set, indicates that your application is to be notified

whenever an application launched by your application terminates or crashes. If the Process Manager is available, your application receives this information as an Apple event, the Application Died event. See the chapter “Process Manager” chapter in *Inside Macintosh: Processes* for more information about launching applications and receiving Application Died events.

Note

Some early versions of MultiFinder do not send application-died events, and your application should not depend on receiving them if it is running in System 6. These events are provided primarily for use by debuggers. ♦

`is32BitCompatible` When set, indicates that your application can be run with the 32-bit Memory Manager. You should not set this flag unless you have thoroughly tested your application on a 32-bit system (such as a Macintosh IIfx computer running System 7 in 32-bit mode or under A/UX). This flag is not used in System 7.

`isHighLevelEventAware` When set, indicates that your application can send and receive high-level events. If this flag is not set, the Event Manager does not give your application high-level events when you call `WaitNextEvent` (page 7-423). There is no way to mask out specific types of high-level events; if this flag is set, your application receives all types of high-level events sent to your application.

Your application must support the four required Apple events if you set the `isHighLevelEventAware` flag. See *Inside Macintosh: Interapplication Communication* for information that describes how to respond to the four required Apple events.

`localAndRemoteHLEvents` When set, indicates that your application is to be visible to applications running on other computers on a network (in addition to applications running on the local computer). If this flag is not set, your application does not receive high-level events across a network.

Event Manager

`isStationeryAware` When set, indicates that your application can recognize stationery documents. If this flag is not set and the user opens a stationery document, the Finder duplicates the document and prompts the user for a name for the duplicate document. For information about how your application can use stationery documents, see the chapter “Finder Interface” in this book.

`useTextEditServices` When set, indicates that your application can use the inline text services provided by TextEdit. See *Inside Macintosh: Text* for information about the inline input capabilities of TextEdit.

The numbers you specify as your application’s preferred and minimum memory sizes depend on the particular memory requirements of your application. Your application’s memory requirements depend on the size of your application’s static heap, dynamic heap, A5 world, and stack. (See “Introduction to Memory Management” in *Inside Macintosh: Memory* for complete details about these areas of your application’s partition.)

The static heap size includes objects that are always present during the execution of the application—for example, code segments, Toolbox data structures for window records, and so on.

Dynamic heap requirements depend on how many objects are created on a per-document basis (which may vary in size proportionally with the document itself) and the number of objects that are required for specific commands or functions.

The size of the A5 world depends on the amount of global data and the number of intersegment jumps the application contains.

Finally, the stack contains variables, return addresses, and temporary information. The application stack size varies among computers, so you should base your values for the stack size according to the stack size required on a Macintosh Plus (8 KB). The Process Manager automatically adjusts your requested amount of memory to compensate for the different stack sizes on different machines. For example, if you request 512 KB, more stack space (approximately 16 KB) will be allocated on machines with larger default stack sizes.

Window Manager

This chapter describes the Window Manager's data structures and functions. It also lists the resources used by the Window Manager and describes the window ('WIND') and window color table ('wctb') resources.

Window Manager Constants and Data Types

This section describes the Window Manager constants, including those used to identify window types, regions of the screen, and tasks performed by a window definition function.

This section also describes the Window Manager data structures and related types: the window record, the color window record, the state data record, the window color table structure, the auxiliary window record, and the window list.

A window record or color window record describes an individual window. It includes the record for the graphics port in which the window is displayed.

The state data record stores two rectangles, known as the user state and the standard state, which define the size and location of the window as specified by the user and by your application. Your application switches between the two states when the user clicks the zoom box.

A window color table defines the colors to be used for drawing the window's frame and highlighting selected text. Ordinarily, you use the default window color table, which produces windows in the colors selected by the user through the Color control panel. If your application has some unusual need to control the frame colors, you can set up your own window color tables.

The Window Manager uses auxiliary window records to associate a window with its window color table.

The Window Manager uses the window list to track all of the windows on the desktop.

Window Definition IDs

The Window Manager supports nine standard window types and eight standard floating window types. You can create windows of the standard types by passing one of the window definition ID enumerators to `NewCWindow` (page 8-503) or `NewWindow` (page 8-506). You can also use these enumerators to specify one of the standard window types in a window resource; see “The Window Resource” (page 8-568) for details.

A window definition ID incorporates both the window’s definition function and its variation code. The resource ID of the window definition function is stored in the upper 12 bits of the integer, and the variation code is stored in the lower 4 bits.

Standard Window Types

These are the window definition IDs for the standard window types:

```
enum {                                /* window definition IDs for standard window types */
    documentProc    = 0,    /* standard document dialog box */
    dBoxProc        = 1,    /* alert box or modal dialog box */
    plainDBox       = 2,    /* plain box */
    altDBoxProc     = 3,    /* plain box with shadow */
    noGrowDocProc   = 4,    /* movable window, no size box or zoom box */
    movableDBoxProc = 5,    /* movable modal dialog box */
    zoomDocProc     = 8,    /* standard document window */
    zoomNoGrow      = 12,   /* zoomable, nonresizable window */
    rDocProc        = 16    /* rounded-corner window */
};
```

Enumerator descriptions

<code>documentProc</code>	Window definition ID for a standard document dialog box.
<code>dBoxProc</code>	Window definition ID for an alert box or modal dialog box.
<code>plainDBox</code>	Window definition ID for a plain dialog box.
<code>altDBoxPro</code>	Window definition ID for a plain dialog box with a shadow.
<code>noGrowDocProc</code>	Window definition ID for a movable window with no size or zoom box.

Window Manager

<code>movableDBoxProc</code>	Window definition ID for a movable modal dialog box.
<code>zoomDocProc</code>	Window definition ID for a standard document window.
<code>zoomNoGrow</code>	Window definition ID for a zoomable, nonresizeable window.
<code>rDocProc</code>	Window definition ID for a rounded-corner window.

You can also add a zoom box to a movable modal dialog box by specifying the sum of two constants: `movableDBoxProc + zoomDocProc`, but a zoom box is not recommended on any dialog box.

You can control the diameter of curvature of a rounded-corner window (window type `rDocProc`) by adding one of these integers to the `rDocProc` constant:

Window definition ID	Diameters of curvature
<code>rDocProc</code>	16, 16
<code>rDocProc + 2</code>	4, 4
<code>rDocProc + 4</code>	6, 6
<code>rDocProc + 6</code>	10, 10

Floating Window Types

These are the window definition IDs for the standard floating window types:

```
enum { /* window definition IDs for standard floating window types */
    floatProc          = 1985,    /* standard floating window */
    floatGrowProc      = 1987,    /* resizable floating window */
    floatZoomProc      = 1989,    /* zoomable floating window */
    floatZoomGrowProc  = 1991,    /* zoomable, resizable floating
                                   /* window */
    floatSideProc      = 1993,    /* sideways floating window,
                                   /* no size or zoom box */
    floatSideGrowProc  = 1995,    /* sideways floating window
                                   /* with grow box */
    floatSideZoomProc  = 1997,    /* sideways floating window
                                   /* with zoom box */
    floatSideZoomGrowProc = 1999  /* sideways floating window
                                   /* with grow box and zoom box */
};
```

Enumerator descriptions

<code>floatProc</code>	Window definition ID for a plain floating window with no size box or zoom box.
<code>floatGrowProc</code>	Window definition ID for a floating window with a grow box.
<code>floatZoomProc</code>	Window definition ID for a floating window with a zoom box.
<code>floatZoomGrowProc</code>	Window definition ID for a floating window with a grow box and a zoom box.
<code>floatSideProc</code>	Window definition ID for a sideways floating window with no size or zoom box.
<code>floatSideGrowProc</code>	Window definition ID for a sideways floating window with a grow box.
<code>floatSideZoomProc</code>	Window definition ID for a sideways floating window with a zoom box.
<code>floatSideZoomGrowProc</code>	Window definition ID for a sideways floating window with a grow and a zoom box.

Resource IDs of Standard Window Definition Functions

These are the resource IDs for the standard window definition functions:

```
enum {          /* resource IDs for standard window definition functions */
    kStandardWindowDefinition    = 0,      /* for document windows and dialogs */
    kRoundWindowDefinition       = 1,      /* DA-style window */
    kFloatingWindowDefinition    = 124     /* for floating windows */
};
```

`kStandardWindowDefinition`
Resource ID for document windows dialog boxes.

`kRoundWindowDefinition`
Resource ID for desk-accessory style windows.

`kFloatingWindowDefinition`
Resource ID for floating windows.

Variant Codes

These are the variant codes used with the standard and floating window definition functions:

```
enum {
/* for use with kStandardWindowDefinition */
    kModalDialogVariantCode          = 1,          /* modal dialog variant code */
    kMovableModalDialogVariantCode    = 5,          /* movable modal dialog
                                                    /* variant code */

/* for use with kFloatingWindowDefinition */
    kSideFloaterVariantCode           = 8           /* sideways floating window
                                                    /* variant code */
};
```

kModalDialogVariantCode

Variant code for modal dialog boxes for use with the standard window definition function.

kMovableModalDialogVariantCode

Variant code for movable modal dialog boxes for use with the standard window definition function.

kSideFloaterVariantCode

Variant code for sideways floating window for use with the floating window definition function.

FindWindow Result Codes

When your application receives a mouse event, you typically call the `FindWindow` function (page 8-517). `FindWindow` returns an integer that specifies where the cursor was when the user pressed the mouse button.

```
enum {
/* FindWindow result codes */
    inDesk          = 0,    /* none of the following */
    inMenuBar        = 1,    /* in menu bar */
    inSysWindow      = 2,    /* in desk accessory window */
    inContent         = 3,    /* anywhere in content rgn except size box
                              /* if window is active, anywhere including
                              /* size box if window is inactive */

    inDrag           = 4,    /* in drag (title bar) region */
    inGrow            = 5,    /* in size box (active window only) */
};
```

Window Manager

```

    inGoAway      = 6,    /* in close box */
    inZoomIn      = 7,    /* in zoom box (window in standard state) */
    inZoomOut     = 8     /* in zoom box (window in user state) */
};

```

Enumerator descriptions

<code>inDesk</code>	<p>The cursor is not in the menu bar, a desk accessory window, or any window that belongs to your application. The <code>FindWindow</code> function (page 8-517) might return this value if, for example, the user presses the mouse button while the cursor is on the window frame but not in the title bar, close box, or zoom box. When <code>FindWindow</code> returns <code>inDesk</code>, your application doesn't need to do anything. In System 7, when the user presses the mouse button while the cursor is on the desktop or in a window that belongs to another application, the Event Manager sends your application a suspend event and switches to the Finder or another application.</p>
<code>inMenuBar</code>	<p>The user has pressed the mouse button while the cursor is in the menu bar. When <code>FindWindow</code> returns <code>inMenuBar</code>, your application typically adjusts its menus and then calls the Menu Manager's function <code>MenuSelect</code> to let the user choose menu items. For more on <code>MenuSelect</code>, see the chapter "Menu Manager Reference" in this book.</p>
<code>inSysWindow</code>	<p>The user has pressed the mouse button while the cursor is in a window belonging to a desk accessory that was launched in your application's partition. This situation seldom arises in System 7. When the user clicks in a window belonging to a desk accessory launched independently, the Event Manager sends your application a suspend event and switches to the desk accessory.</p> <p>If <code>FindWindow</code> does return <code>inSysWindow</code>, your application calls the <code>SystemClick</code> function, described in <i>Inside Macintosh: Macintosh Toolbox Essentials</i> (page 7-435). The <code>SystemClick</code> function routes the event to the desk accessory. If the user presses the mouse button with the cursor in the content region of an inactive desk accessory window, <code>SystemClick</code> makes the window active by sending your application and the desk accessory the appropriate activate events.</p>

Window Manager

<code>inContent</code>	The user has pressed the mouse button while the cursor is in the content area (excluding the size box in an active window) of one of your application's windows. When <code>FindWindow</code> (page 8-517) returns <code>inContent</code> , your application calls its function for handling clicks in the content region.
<code>inDrag</code>	The user has pressed the mouse button while the cursor is in the drag region of a window (that is, the title bar, excluding the close box and zoom box). When <code>FindWindow</code> returns <code>inDrag</code> , your application calls the Window Manager's <code>DragWindow</code> function (page 8-519) to let the user drag the window to a new location.
<code>inGrow</code>	The user has pressed the mouse button while the cursor is in an active window's size box. When <code>FindWindow</code> returns <code>inGrow</code> , your application calls its own function for resizing a window.
<code>inGoAway</code>	The user has pressed the mouse button while the cursor is in an active window's close box. When <code>FindWindow</code> returns <code>inGoAway</code> , your application calls the <code>TrackGoAway</code> function (page 8-531) to track mouse activity while the button is down and then calls its own function for closing a window if the user releases the button while the cursor is in the close box.
<code>inZoomIn</code> or <code>inZoomOut</code>	The user has pressed the mouse button while the cursor is in an active window's zoom box. When <code>FindWindow</code> returns <code>inZoomIn</code> or <code>inZoomOut</code> , your application calls the <code>TrackBox</code> function (page 8-529) to track mouse activity while the button is down and then calls its own function for zooming a window if the user releases the button while the cursor is in the zoom box.

Window Kinds

When it creates a window, the Window Manager places a value in the `windowKind` field of a window record that identifies how the window was created; see “The Color Window Record” (page 8-483) for details. The `windowKind` field typically contains one of these values:

Window Manager

```
enum {                                /* new names for dialog kind enumerators */
    kDialogWindowKind      = 2,      /* dialog or alert window */
    kApplicationWindowKind = 8       /* window created by an
                                       /* application */
};

enum {                                /* old names for dialog kind enumerators */
    dialogKind      = 2,      /* dialog window */
    userKind        = 8       /* user window */
};
```

kDialogWindowKind

Identifies all dialog or alert box windows, whether created by system software or, indirectly through the Dialog Manager, by your application. The Dialog Manager uses this field to help it track dialog and alert box windows.

kApplicationWindowKind

Identifies a window created directly by your application.

Part Identifiers for ColorSpec Records

To specify the color for part of a window, you use a `ColorSpec` record in the `ctTable` field of a window color table record; see “The Window Color Table Record” (page 8-490) for details. To specify the window part, you specify one of these values in the `partIdentifier` field of the `ColorSpec` record:

```
enum {
    wContentColor      = 0,      /* content rgn background */
    wFrameColor        = 1,      /* window outline */
    wTextColor          = 2,      /* window title and
                                   /* button text */
    wHiliteColor        = 3,      /* reserved */
    wTitleBarColor      = 4,      /* reserved */
    wHiliteColorLight   = 5,      /* lightest stripes in
                                   /* title bar & lightest
                                   /* dimmed text */
    wHiliteColorDark    = 6,      /* darkest stripes in
                                   /* title bar and
                                   /* darkest dimmed text */
    wTitleBarLight      = 7,      /* lightest parts of
```

Window Manager

```

                                /* title bar background */
wTitleBarDark                = 8,  /* darkest parts of */
                                /* title bar background */
wDialogLight                 = 9,  /* lightest element
                                /* of dialog box frame */
wDialogDark                  = 10, /* darkest element of
                                /* dialog box frame */
wTingeLight                  = 11, /* lightest window tinging */
wTingeDark                   = 12  /* darkest window tinging */
};

```

Enumerator descriptions

wContentColor	Produces background color for content region of window.
wFrameColor	Produces color of window's outline.
wTextColor	Produces color of window's title and button text.
wHiliteColor	Reserved.
wTitleBarColor	Reserved.
wHiliteColorLight	Produces color of the lightest stripes in the title bar and the lightest dimmed text.
wHiliteColorDark	Produces color of the darkest stripes in the title bar and the darkest dimmed text.
wTitleBarLight	Produces color of the lightest parts of the title bar background.
wTitleBarDark	Produces color of the darkest parts of the title bar background.
wDialogLight	Produces color of the lightest element of the dialog box frame.
wDialogDark	Produces color of the darkest element of the dialog box frame.
wTingeLight	Produces color of the lightest window tinging.
wTingeDark	Produces color of the darkest window tinging.

The Color Window Record

The Window Manager maintains a window record or color window record for each window on the desktop.

The Window Manager supplies functions that let you access the window record as necessary. Your application seldom changes fields in the window record directly.

The `CWindowRecord` data type defines the window record for a color window. The `CWindowPeek` data type is a pointer to a color window record. The first field in the window record is in fact the record that describes the window's graphics port. The `CWindowPtr` data type is defined as a `CGrafPtr`, which is a pointer to a graphics port (in this case, the window's graphics port).

When Color QuickDraw is not available, you can create monochrome windows using the parallel data types `WindowRecord`, `WindowPeek`, and `WindowPtr`, described in “The Window Record” (page 8-488).

For compatibility, the `WindowPtr` and `WindowPeek` data types can point to either a color window record or a monochrome window record. You use the `WindowPtr` data type to specify a window in most Window Manager functions, and you can use it to specify a graphics port in QuickDraw functions that take the `GrafPtr` data type. Note that you can access only the fields of the window's graphics port, not the rest of the window record, through the `WindowPtr` and `CWindowPtr` data types. You use the `WindowPeek` and `CWindowPeek` data types in low-level Window Manager functions and in your own functions that access window record fields beyond the graphics port.

The functions that manipulate color windows get color information from the window color tables and the auxiliary window record described in the sections “The Window Color Table Record” (page 8-490) and “The Auxiliary Window Record” (page 8-492).

```
struct WindowRecord {
    GrafPort      port;           /* window's graphics port */
    short         windowKind;    /* class of the window */
    Boolean       visible;       /* visibility */
    Boolean       hilited;       /* highlighting */
    Boolean       goAwayFlag;    /* presence of close box */
    Boolean       spareFlag;     /* presence of zoom box */
    RgnHandle     strucRgn;      /* handle to structure
                                /* region */
    RgnHandle     contrRgn;      /* handle to content
                                /* region */
    RgnHandle     updateRgn;     /* handle to update region */
    Handle        windowDefProc; /* handle to window
                                /* definition function */
}
```

Window Manager

```

        Handle          dataHandle;          /* handle to window state
                                                /* data record */

        StringHandle    titleHandle;          /* handle to window title */
        short           titleWidth;           /* title width in pixels */
        ControlHandle    controlList;          /* handle to window's
                                                /* control list */

        CWindowPeek     nextWindow;           /* next window in window
                                                /* list */

        PicHandle        windowPic;           /* handle to optional
                                                /* picture */

        long             refCon;              /* reference constant */
};
typedef struct CWindowRecord CWindowRecord;
typedef CWindowRecord *CWindowPeek;
typedef CGrafPtr CWindowPtr;

```

Field descriptions

port	<p>The graphics port record that describes the graphics port in which the window is drawn.</p> <p>The graphics port record, which is documented in <i>Inside Macintosh: Imaging</i>, defines the rectangle in which drawing can occur, the window's visible region, the window's clipping region, and a collection of current drawing characteristics such as fill pattern, pen location, and pen size.</p>
windowKind	<p>The class of window—that is, how the window was created.</p> <p>The Window Manager fills in this field when it creates the window record. It places a negative value in <code>windowKind</code> when the window was created by a desk accessory. (The value is the reference ID of the desk accessory.) This field can also contain one of two values described in “Window Kinds” (page 8-481) indicating how the window was created: <code>dialogKind</code> or <code>userKind</code>.</p>
visible	<p>A Boolean value indicating whether or not the window is visible. If the window is visible, the Window Manager sets this field to <code>true</code>; if not, <code>false</code>. Visibility means only whether or not the window is to be displayed, not necessarily whether you can see it on the screen. (For example, a window that is completely covered by other</p>

	windows can still be visible, even if the user cannot see it on the screen.)
<code>hilited</code>	A Boolean value indicating whether the window is highlighted—that is, drawn with stripes in the title bar. Only the active window is ordinarily highlighted. When the window is highlighted, the <code>hilited</code> field contains <code>true</code> ; when not, <code>false</code> .
<code>goAwayFlag</code>	<p>A Boolean value indicating whether the window has a close box.</p> <p>The Window Manager fills in this field when it creates the window according to the information in the 'WIND' resource or the parameters passed to the function that creates the window.</p> <p>If the value of <code>goAwayFlag</code> is <code>true</code>, and if the window type supports a close box, the Window Manager draws a close box when the window is highlighted.</p>
<code>spareFlag</code>	A Boolean value indicating whether the window type supports zooming. The Window Manager sets this field to <code>true</code> if the window's type is one that includes a zoom box (<code>zoomDocProc</code> , <code>zoomNoGrow</code> , or even <code>modalDBoxProc</code> + <code>zoomDocProc</code>).
<code>strucRgn</code>	A handle to the structure region, which is defined in global coordinates. The structure region is the entire screen area covered by the window—that is, both the window contents and the window frame.
<code>contRgn</code>	A handle to the content region, which is defined in global coordinates. The content region is the part of the window that contains the document, dialog, or other data; the window controls; and the size box.
<code>updateRgn</code>	A handle to the update region, which is defined in global coordinates. The update region is the portion of the window that must be redrawn. It is maintained jointly by the Window Manager and your application. The update region excludes parts of the window that are covered by other windows.
<code>windowDefProc</code>	A handle to the definition function that controls the window.

There's no need for your application to access this field directly.

In Macintosh models that use only 24-bit addressing, this field contains both a handle to the window's definition function and the window's variation code. If you need to know the variation code, regardless of the addressing mode, call the `GetWVariant` function (page 8-540).

`dataHandle`

Usually a handle to a data area used by the window definition function.

For zoomable windows, `dataHandle` contains a handle to the `WStateData` record, which contains the user state and standard state rectangles. The `WStateData` record is described in "The Window State Data Record" (page 8-489).

A window definition function that needs only 4 bytes of data can use the `dataHandle` field directly, instead of storing a handle to the data. The window definition function that handles rounded-corner windows, for example, stores the diameters of curvature in the `dataHandle` field.

`titleHandle`

A handle to the string that defines the title of the window.

`titleWidth`

The width, in pixels, of the window's title.

`controlList`

A handle to the window's control list, which is used by the Control Manager. (See the chapter "Control Manager Reference" in this book for a description of control lists.)

`nextWindow`

A pointer to the next window in the window list, that is, the window behind this window on the desktop. In the window record for the last window on the desktop, the `nextWindow` field is set to `nil`.

`windowPic`

A handle to a QuickDraw picture of the window's contents. The Window Manager initially sets the `windowPic` field to `nil`. If you're using the window to display a stable image, you can use the `SetWindowPic` function (page 8-538) to place a handle to the picture in this field. When the window's contents need updating, the Window Manager then redraws the contents itself instead of generating an update event.

`refCon`

The window's reference value field, which is simply storage space available to your application for any purpose.

The sample code in this chapter uses the `refCon` field to associate a window with the data it displays by storing a window type constant in the `refCon` field of alert and dialog window records and a handle to a document record in the `refCon` field of a document window record.

Note

The close box, drag region, zoom box, and size box are not included in the window record because they don't necessarily have the formal data structure for regions as defined in QuickDraw. The window definition function determines where these regions are. ♦

The Window Record

If Color QuickDraw is not available, you create windows with a parallel data structure, the window record. The only difference between a color window record and a window record is that a color window record points to a color graphics port, which allows full use of Macintosh computers with color capability, and a window record points to a monochrome graphics port

The data types that describe window records, `WindowRecord`, `WindowPtr`, and `WindowPeek`, are parallel to the data types that describe color window records, and the fields in the monochrome window record are identical to the fields in the color window record. For a complete description, see “The Color Window Record” (page 8-483).

```
struct WindowRecord {
    GrafPort    port;           /* all fields have same use
                                /* as in color window record */
    short       windowKind;    /*window's graphics port */
    Boolean     visible;       /*class of the window */
    Boolean     hilited;       /* visibility */
    Boolean     hilited;       /* highlighting */
    Boolean     goAwayFlag;    /* presence of close box */
    Boolean     spareFlag;     /* presence of zoom box */
    RgnHandle   strucRgn;      /* handle to structure
                                /* region */
    RgnHandle   contrRgn;      /* handle to content
                                /* region */
    RgnHandle   updateRgn;     /* handle to update region */
    Handle      windowDefProc; /* handle to window
```


Window Manager

```

                                /* definition function */
Handle          dataHandle;    /* *handle to window state
                                /* data record */
StringHandle    titleHandle;   /* handle to window title */
short           titleWidth;    /* title width in pixels */
ControlHandle   controlList;   /* handle to window's
                                /* control list */
WindowPeek      nextWindow;    /* next window in window
                                /* list */
PicHandle       windowPic;     /*handle to optional
                                /* picture */
long            refCon;        /*reference constant */
};
typedef struct WindowRecord WindowRecord;
typedef WindowRecord *WindowPeek;
typedef GrafPtr WindowPtr;

```

The Window State Data Record

The zoom box allows the user to alternate quickly between two window positions and sizes: the user state and the standard state. The Window Manager stores the user state and your application stores the standard state in the window state data record, whose handle appears in the `dataHandle` field of the window record.

The `WStateData` record data type defines the window state data record.

```

struct WStateData {
    Rect    userState;          /* size and location established by user */
    Rect    stdState;           /* size and location established by app */
};
typedef struct WStateData WStateData;
typedef WStateData *WStateDataPtr, **WStateDataHandle;

```

Field descriptions

`userState`

A rectangle that describes the window size and location established by the user.

The Window Manager initializes the user state to the size and location of the window when it is first displayed, and then updates the `userState` field whenever the user resizes a window. Although the user state specifies both the size

`stdState`

and location of the window, the Window Manager updates the state data record only when the user resizes a window—not when the user merely moves a window.

The rectangle describing the window size and location that your application considers the most convenient, considering the function of the document, the screen space available, and the position of the window in its user state. 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. The user cannot change a window’s standard state.

Your application typically calculates and sets the standard state each time the user zooms to the standard state. 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 through Page Setup, the application might adjust the standard state to reflect the new page size. (See *Macintosh Human Interface Guidelines* for a detailed description of how your application determines where to open and zoom windows.)

The `ZoomWindow` function changes the size of a window according to the values in the window state data record. The function changes the window to the user state when the user zooms “in” and to the standard state when the user zooms “out.” For descriptions of the functions you call when zooming windows, see “Zooming Windows” (page 8-528).

The Window Color Table Record

The user controls the colors used for the window frame and text highlighting through the Color control panel. Ordinarily, your application doesn’t override the user’s color choices, which are stored in a default window color table. If you have some extraordinary need to control window colors, you can do so by defining window color tables for your application’s windows.

The Window Manager maintains window color information tables in a data structure of type `WinCTab`.

You can define your own window color table and apply it to an existing window through the `SetWinColor` function (page 8-544).

To establish the window color table for a window when you create it, you provide a window color table ('wctb') resource with the same resource ID as the 'WIND' resource that defines the window.

The `WCTabPtr` data type is a pointer to a window color table record, and the `WTabHandle` is a handle to a window color table record. The `WinCTab` data type defines a window color table record.

```
struct WinCTab {
    long          wCSeed;          /* reserved */
    short         wCReserved;      /* reserved */
    short         ctSize;          /* number of entries in table -1*/
    ColorSpec     ctTable[5];      /* array of color specification */
                                   /* records */
};
typedef struct WinCTab WinCTab;
typedef WinCTab *WCTabPtr, **WCTabHandle;
```

Field descriptions

<code>wCSeed</code>	Reserved.
<code>wCReserved</code>	Reserved.
<code>ctSize</code>	The number of entries in the table, minus 1. If you're building a color table for use with the standard window definition function, the maximum value of this field is 12. Custom window definition functions can use color tables of any size.
<code>ctTable</code>	An array of <code>colorSpec</code> records. In a window color table, each <code>colorSpec</code> record specifies a window part in the first word and an RGB value in the other three words:

```
struct ColorSpec {
    short         value;          /* part identifier */
    RGBColor      rgb;            /* RGB value */
};
```

The `value` field of a `colorSpec` record specifies a constant that defines which part of the window the color controls. For the window color table used by the standard window

definition function, you can specify one of the values listed in “Part Identifiers for ColorSpec Records” (page 8-482).

Note

The part codes in System 5 and System 6 are significantly different from the part codes described here, which apply only to System 7. ♦

The window parts can appear in any order in the table.

The `rgb` field of a `ColorSpec` record contains three words of data that specify the red, green, and blue values of the color to be used. The `RGBColor` data type is defined in *Inside Macintosh: Imaging*.

When your application creates a window, the Window Manager first looks for a resource of type `'wctb'` with the same resource ID as the `'WIND'` resource used for the window. If it finds one, it creates a window color table for the window from the information in that resource, and then displays the window in those colors. If it doesn't find a window color table resource with the same resource ID as your window resource, the Window Manager uses the default system window color table, read into the heap during application startup.

After creating a window, you can change the entries in a window's window color table with the `SetWinColor` function (page 8-544).

See “The Window Color Table Resource” (page 8-571) for a description of the window color table resource.

The Auxiliary Window Record

The auxiliary window record specifies the color table used by a window and contains reference information used by the Dialog Manager and the Window Manager.

The Window Manager creates and maintains the information in an auxiliary window record; your application seldom, if ever, needs to access an auxiliary window record.

```
struct AuxWinRec {
    AuxWinHandle    awNext;           /* handle to next record */
    WindowPtr       awOwner;         /* pointer to window
                                     /* associated with this
                                     /* record */
```

Window Manager

```

    CTabHandle    awCTable;        /* handle to color table */
    Handle        dialogCItem;    /* storage used by
                                   /* Dialog Manager */

    long          awFlags;        /* reserved */
    CTabHandle    awReserved;     /* reserved */
    long          awRefCon;       /* reference constant, for
                                   /* use by application */
};
typedef struct AuxWinRec AuxWinRec;
typedef AuxWinRec *AuxWinPtr, **AuxWinHandle;

```

Field descriptions

<code>awNext</code>	A handle to the next record in the auxiliary window list, used by the Window Manager to maintain the auxiliary window list as a linked list. If a window is using the default auxiliary window record, this value is <code>nil</code> .
<code>awOwner</code>	A pointer to the window that uses this record. The <code>awOwner</code> field of the default auxiliary window record is set to <code>nil</code> .
<code>awCTable</code>	A handle to the window's color table. Unless you specify otherwise, this is a handle to the system window color table.
<code>dialogCItem</code>	Private storage for use by the Dialog Manager.
<code>awFlags</code>	Reserved.
<code>awReserved</code>	Reserved.
<code>awRefCon</code>	The reference constant, typically used by an application to associate the auxiliary window record with a document record.

Except in unusual circumstances, your application doesn't need to manipulate window color tables or the auxiliary window record.

For compatibility with other applications in the shared environment, your application should not manipulate system color tables directly but should go through the Palette Manager, documented in *Inside Macintosh: Imaging*. If your application provides its own window and control definition functions, these functions should apply the user's desktop color choices the same way the standard window and control definition functions do.

The Window List

The Window Manager maintains information about the windows on the desktop in a private structure called the *window list*. The window list contains pointers to all windows on the desktop, both visible and invisible, and contains other information that the Window Manager uses to maintain the desktop.

Your application should not directly access the information in a window list. The structure of the window list is private to the Window Manager.

The global variable `WindowList` contains a pointer to the first window in the window list. If you need this pointer, do not access it directly. Instead, use the accessors `LMSetWindowList` (page 8-561) and `LMGetWindowList` (page 8-562).

Window Definition Function Type and Macros

The Window Manager calls your window definition function when it needs to draw any specialized window types you have defined for your application. The Window Manager supplies window definition functions that handle the standard window types.

The Window Manager defines a window definition function as follows:

```
typedef pascal long (*WindowDefProcPtr)(
    short varCode,
    WindowPtr theWindow,
    short message,
    long param);
```

For information about writing a window definition function, see “Application-Defined Function” (page 8-562).

Note

Since you normally store a window definition function in a resource, you don’t typically have to deal with it directly; the Window Manager loads it and calls it when necessary. Therefore, most programmers don’t need the information in the remainder of this section. ♦

The Window Manager defines the universal procedure pointer `WindowDefUPP` for identifying a window definition function:

```
typedef UniversalProcPtr WindowDefUPP;
```

Window Manager

The Window Manager defines the macro `NewWindowDefProc` for obtaining a `WindowDefUPP`:

```
#define NewWindowDefProc(userRoutine)\
    (WindowDefUPP) NewRoutineDescriptor((ProcPtr)(userRoutine),\
    uppWindowDefProcInfo, GetCurrentArchitecture())
```

The Window Manager also defines the macro `CallWindowDefProc` for calling a `WindowDefUPP`:

```
#define CallWindowDefProc(userRoutine, varCode, theWindow, message, param)\
    CallUniversalProc((UniversalProcPtr)(userRoutine), uppWindowDefProcInfo,\
    (varCode), (theWindow), (message), (param))
```

For more information about universal procedure pointers and the Mixed Mode Manager, see *Inside Macintosh: PowerPC System Software*.

Window Definition Function Enumerators

The Window Manager passes one of the following values in the `message` parameter of a window definition function:

```
enum {
    wDraw          = 0,    /* draw window frame */
    wHit           = 1,    /* report where mouse-down event occurred */
    wCalcRgns      = 2,    /* calculate strucRgn and contRgn */
    wNew           = 3,    /* perform additional initialization */
    wDispose       = 4,    /* perform additional disposal */
    wGrow          = 5,    /* draw grow image during resizing */
    wDrawGIcon     = 6,    /* draw size box and scroll bar outline */
};
```

Enumerator descriptions

<code>wDraw</code>	Draws the window's frame.
<code>wHit</code>	Reports the location of a mouse-down event.
<code>wCalcRgns</code>	Calculates the structure region and the content region.
<code>wNew</code>	Performs additional initialization.
<code>wDispose</code>	Performs additional disposal.

Window Manager

`wGrow` Draws the grow image during resizing operation.

`wDrawGIcon` Draws the outlines for the size box and the scroll bar.

If the window definition function receives a `wHit` message, it should return one of these values:

```
enum {
    wNoHit      = 0,    /* none of the following */
    wInContent  = 1,    /* in content region (except grow, if active) */
    wInDrag     = 2,    /* in drag region */
    wInGrow     = 3,    /* in grow region (active window only) */
    wInGoAway   = 4,    /* in go-away region (active window only) */
    wInZoomIn   = 5,    /* in zoom box for zooming in (active window
                        /* only) */
    wInZoomOut  = 6,    /* in zoom box for zooming out (active window
                        /* only) */
};
```

Enumerator descriptions

`wNoHit` Mouse-down did not occur in the content region or the drag region of any window or the grow, go-away or zoom box for zooming in or out of an active window.

`wInContent` Mouse-down occurred in the content region with the exception of the grow region if the window is active.

`wInDrag` Mouse-down occurred in the drag region.

`wInGrow` Mouse-down occurred in the grow region of an active window.

`wInGoAway` Mouse-down occurred in the go-away region of an active window

`wInZoomIn` Mouse-down occurred in the in-zoom box for zooming in an active window.

`wInZoomOut` Mouse-down occurred in the out-zoom box for zooming out of an active window.

For information about writing a window definition function, see “Application-Defined Function” (page 8-562).

Window Manager Functions

This section describes the complete set of functions for creating, displaying, and managing windows.

Initializing the Window Manager

Before using any other Window Manager functions, you must initialize the Window Manager by calling the `InitWindows` function (page 8-497).

As part of initialization, `InitWindows` creates the **Window Manager port**, a graphics port that occupies all of the main screen. The Window Manager port is named `WMgrCPort` on Macintosh computers equipped with Color QuickDraw and `WMgrPort` on computers with only QuickDraw.

Ordinarily, your application does not need to know about the Window Manager port. If necessary, however, you can retrieve a pointer to it by calling the function `GetWMgrPort` (page 8-543) or `GetCWMgrPort` (page 8-543). Your application should not draw directly into the Window Manager port, except through custom window definition functions.

The Window Manager draws your application's windows into the Window Manager port. The port rectangle of the Window Manager port is the bounding rectangle of the main screen (`screenBits.bounds`). To accommodate systems with multiple monitors, QuickDraw recognizes a port rectangle of `screenBits.bounds` as a special case and allows drawing on all parts of the desktop.

InitWindows

Initializes the Window Manager for your application.

```
pascal void InitWindows(void);
```

DESCRIPTION

Before calling `InitWindows`, you must initialize QuickDraw and the Font Manager by calling the `InitGraf` and `InitFonts` routines, documented in *Inside Macintosh: Imaging* and *Inside Macintosh: Text*.

ASSEMBLY-LANGUAGE INFORMATION

When the desktop needs to be redrawn any time after initialization, the Window Manager checks the global variable `DeskHook`, which can be used as a pointer to an application-defined function for drawing the desktop. This variable is ordinarily set to 0, but not until after system startup. If you're displaying windows in code that is to be executed during startup, set `DeskHook` to 0. Note that the use of the Window Manager's global variables is not guaranteed to be compatible in system software versions later than System 6.

Creating Windows

You can create windows in two ways:

- from a window resource (a resource of type 'WIND'), with the `GetNewCWindow` (page 8-498) and `GetNewWindow` (page 8-501) functions
- from a collection of window characteristics passed as parameters to the `NewCWindow` and `NewWindow` (page 8-506) functions

Creating windows from resources allows you to localize your application for different languages and to change the characteristics of your windows during application development by changing only the window resources.

All four functions, `GetNewCWindow` (page 8-498), `GetNewWindow` (page 8-501), `NewCWindow` (page 8-503), and `NewWindow` (page 8-506), can allocate space in your application's heap for the new window's window record. For more control over memory use, you can allocate the space yourself and pass a pointer when creating a window. In either case, the Window Manager fills in the data structure and returns a pointer to it.

GetNewCWindow

Creates a color window with the properties defined in the 'WIND' resource with a specified resource ID.

```
pascal WindowPtr GetNewCWindow (short windowID,
                                void *wStorage,
                                WindowPtr behind);
```

<code>windowID</code>	The resource ID of the 'WIND' resource that defines the properties of the window.
<code>wStorage</code>	<p>A pointer to memory space for the window record.</p> <p>If you specify a value of <code>nil</code> for <code>wStorage</code>, the <code>GetNewCWindow</code> function allocates the window record as a nonrelocatable object in the heap. You can reduce the chances of heap fragmentation by allocating the memory your application needs for window records early in your initialization code. Whenever you need to create a window, you can allocate memory from your own block and pass a pointer to it in the <code>wStorage</code> parameter.</p>
<code>behind</code>	<p>A pointer to the window that appears immediately in front of the new window on the desktop.</p> <p>To place a new window in front of all other windows on the desktop, specify a value of <code>(WindowPtr)-1L</code>. When you place a window in front of all others, <code>GetNewCWindow</code> removes the highlighting from the previously active window, highlights the newly created window, and generates the appropriate activate events. Note that if you create an invisible window in front of all others on the desktop, the user sees no active window until you make the new window visible (or make another window active).</p> <p>To place a new window behind all other windows, specify a value of <code>nil</code>.</p>

DESCRIPTION

The `GetNewCWindow` function creates a new color window from the specified window resource and returns a pointer to the newly created window record. You can use the returned window pointer to refer to this window in most Window Manager functions. If `GetNewCWindow` is unable to read the window or window definition function from the resource file, it returns `nil`.

The `GetNewCWindow` function looks for a 'wctb' resource with the same resource ID as that of the 'WIND' resource. If it finds one, it uses the window color information in the 'wctb' resource for coloring the window frame and highlighting selected text.

If the window's definition function (specified in the window resource) is not already in memory, `GetNewCWindow` reads it into memory and stores a handle to it in the window record. It allocates space in the application heap for the structure

and content regions of the window and asks the window definition function to calculate those regions.

To create the window, `GetNewCWindow` retrieves the window characteristics from the window resource and then calls the `NewCWindow` function, passing the characteristics as parameters.

The `GetNewCWindow` function creates a window in a color graphics port. Before calling `GetNewCWindow`, verify that Color QuickDraw is available. Your application typically sets up its own global variables reflecting the system setup during initialization by calling the `Gestalt` function. See *Inside Macintosh: Overview* for more information about establishing the local configuration.

SPECIAL CONSIDERATIONS

Note that the `GetNewCWindow` function returns a value of type `WindowPtr`, not `CWindowPtr`.

If you let the Window Manager create the window record in your application's heap, call `DisposeWindow` (page 8-533) to dispose of the window's window record. If you allocated the memory for the window record yourself and passed a pointer to the storage to `GetNewCWindow`, use the `CloseWindow` function (page 8-532) to close the window and the Memory Manager routine `DisposePtr` to dispose of the window record.

SEE ALSO

For more information about window characteristics and the window resource, see `NewCWindow` (page 8-503) and “The Window Resource” (page 8-568).

For more information about closing a window and removing the structures from memory, see the descriptions of the `DisposeWindow` function (page 8-533), the `CloseWindow` function (page 8-532), and the `DisposePtr` function (documented in *Inside Macintosh: Memory*).

GetNewWindow

Creates a new window from a window resource when Color QuickDraw is not available.

```
pascal WindowPtr GetNewWindow (short windowID,
                                void *wStorage,
                                WindowPtr behind);
```

windowID The resource ID of the 'WIND' resource that defines the properties of the window.

wStorage A pointer to memory space for the window record.

If you specify a value of `nil` for `wStorage`, the `GetNewWindow` function allocates the window record as a nonrelocatable object in the heap. You can reduce the chances of heap fragmentation by allocating the memory your application needs for window records early in your initialization code. Whenever you need to create a window, you can allocate memory from your own block and pass a pointer to it in the `wStorage` parameter.

behind A pointer to the window that appears immediately in front of the new window on the desktop.

To place a new window in front of all other windows on the desktop, specify a value of `(WindowPtr)-1L`. When you place a window in front of all others, `GetNewWindow` removes the highlighting from the previously active window, highlights the newly created window, and generates the appropriate activate events. Note that if you create an invisible window in front of all others on the desktop, the user sees no active window until you make the new window visible (or make another window active).

To place a new window behind all other windows, specify a value of `nil`.

DESCRIPTION

The `GetNewWindow` function takes the same parameters as `GetNewCWindow` (page 8-498) and returns a value of type `WindowPtr`. The only difference is that it creates a monochrome graphics port, not a color graphics port. The window record and graphics port record that describe monochrome and color graphics

ports are the same size and can be used interchangeably in most Window Manager functions.

The `GetNewWindow` function creates a new window from the specified window resource and returns a pointer to the newly created window record. You can use the returned window pointer to refer to this window in most Window Manager functions. If `GetNewWindow` is unable to read the window or window definition function from the resource file, it returns `nil`.

If the window's definition function (specified in the window resource) is not already in memory, `GetNewWindow` reads it into memory and stores a handle to it in the window record. It allocates space in the application heap for the structure and content regions of the window and asks the window definition function to calculate those regions.

To create the window, `GetNewWindow` retrieves the window characteristics from the window resource and then calls the function `NewWindow`, passing the characteristics as parameters.

SPECIAL CONSIDERATIONS

If you let the Window Manager create the window record in your application's heap, call `DisposeWindow` (page 8-533) to dispose of the window's window record. If you allocated the memory for the window record yourself and passed a pointer to the storage to `GetNewWindow`, use the `CloseWindow` function (page 8-532) to close the window and the `DisposePtr` routine (documented in *Inside Macintosh: Memory*) to dispose of the window record.

SEE ALSO

For more information about window characteristics and the window resource, see the descriptions of the `NewWindow` function (page 8-506) and "The Window Resource" (page 8-568).

For more information about closing a window and removing the structures from memory, see the descriptions of the `DisposeWindow` function (page 8-533), the `CloseWindow` function (page 8-532), and the `DisposePtr` routine (documented in *Inside Macintosh: Memory*).

NewCWindow

Creates a window with a specified list of characteristics.

```
pascal WindowPtr NewCWindow (void *wStorage,
                             const Rect *boundsRect,
                             ConstStr255Param title,
                             Boolean visible,
                             short procID,
                             WindowPtr behind,
                             Boolean goAwayFlag,
                             long refCon);
```

wStorage A pointer to the window record. If you specify `nil` as the value of `wStorage`, `NewCWindow` allocates the window record as a nonrelocatable object in the application heap. You can reduce the chances of heap fragmentation by allocating memory from a block of memory reserved for this purpose by your application and passing a pointer to it in the `wStorage` parameter.

boundsRect A pointer to a rectangle, given in global coordinates, that specifies the window's initial size and location. This rectangle becomes the port rectangle of the window's graphics port. For the standard window types, the `boundsRect` field defines the content region of the window. The `NewCWindow` function places the origin of the local coordinate system at the upper-left corner of the port rectangle.

Note

The `NewCWindow` function actually calls the `QuickDraw` function `OpenCPort` to create the graphics port. The bitmap, pen pattern, and other characteristics of the window's graphics port are the same as the default values set by `OpenCPort`, except for the character font, which is set to the application font instead of the system font. ♦

title A string that specifies the window's title.

If the title is too long to fit in the title bar, the title is truncated. If the window has a close box, characters are truncated at the end of the title; if there's no close box, the title is centered and truncated at both ends.

To suppress the title in a window with a title bar, pass an empty string, not `nil`, in the title parameter.

`visible`

A Boolean value indicating visibility status: `true` means that the Window Manager displays the window; `false` means it does not.

If the value of the `visible` parameter is `true`, the Window Manager draws a new window as soon as the window exists. The Window Manager first calls the window definition function to draw the window frame. If the value of the `goAwayFlag` parameter is also `true` and the window is frontmost (that is, if the value of the `behind` parameter is `(WindowPtr)-1L`), the Window Manager instructs the window definition function to draw a close box in the window frame. After drawing the frame, the Window Manager generates an update event to trigger your application's drawing of the content region.

When you create a window, you typically specify `false` as the value of the `visible` parameter. When you're ready to display the window, you call the `ShowWindow` function (page 8-513).

`procID`

The window's definition ID, a value that specifies both the window definition function and the variation code within that definition function. For a list of possible values, see "Window Definition IDs" (page 8-476).

`behind`

A pointer to the window that appears immediately in front of the new window on the desktop.

To place a new window in front of all other windows on the desktop, specify a value of `(WindowPtr)-1L`. When you place a new window in front of all others, `NewCWindow` removes highlighting from the previously active window, highlights the newly created window, and generates activate events that trigger your application's updating of both windows. Note that if you create an invisible window in front of all others on the desktop, the user sees no active window until you make the new window visible (or make another window active).

To place a new window behind all other windows, specify a value of `nil`.

Window Manager

<code>goAwayFlag</code>	A Boolean value that determines whether the window has a close box. If the value of <code>goAwayFlag</code> is <code>true</code> and the window type supports a close box, the Window Manager draws a close box in the title bar and recognizes mouse clicks in the close region; if the value of <code>goAwayFlag</code> is <code>false</code> or the window type does not support a close box, it does not.
<code>refCon</code>	The window's reference constant, set and used only by your application.

DESCRIPTION

The `NewCWindow` function creates a window as specified by its parameters, adds it to the window list, and returns a pointer to the newly created window record. You can use the returned window pointer to refer to this window in most Window Manager functions. If `NewCWindow` is unable to read the window definition function from the resource file, it returns `nil`.

The `NewCWindow` function looks for a 'wctb' resource with the same resource ID as the 'WIND' resource. If it finds one, it uses the window color information in the 'wctb' resource for coloring the window frame and highlighting.

If the window's definition function is not already in memory, `NewCWindow` reads it into memory and stores a handle to it in the window record. It allocates space for the structure and content regions of the window and asks the window definition function to calculate those regions.

Storing the characteristics of your windows as resources, especially window titles and window items, makes your application easier to localize.

The `NewCWindow` function creates a window in a color graphics port. Creating color windows whenever possible ensures that your windows appear on color monitors with whatever color options the user has selected. Before calling `NewCWindow`, verify that Color QuickDraw is available. Your application typically sets up its own set of global variables reflecting the system setup during initialization by calling the `Gestalt` function. See the chapter *Inside Macintosh: Overview* for more information about establishing the local configuration.

Note that the function `NewCWindow` returns a value of type `WindowPtr`, not `CWindowPtr`.

SPECIAL CONSIDERATIONS

If you let the Window Manager create the window record in your application's heap, call `DisposeWindow` (page 8-533) to close the window and dispose of its window record. If you allocated the memory for the window record yourself and passed a pointer to `NewCWindow`, use the `CloseWindow` function (page 8-532) to close the window and the `DisposePtr` function (documented in *Inside Macintosh: Memory*) to dispose of the window record.

SEE ALSO

For more information about closing a window and removing the structures from memory, see the descriptions of the `DisposeWindow` function (page 8-533), the `CloseWindow` function (page 8-532), and the `DisposePtr` routine (documented in *Inside Macintosh: Memory*). The `Gestalt` function is described in the chapter “Gestalt Manager” of *Inside Macintosh: Operating System Utilities*.

NewWindow

Creates a new window with the characteristics specified by a list of parameters when Color QuickDraw is not available.

```
pascal WindowPtr NewWindow (void *wStorage,
                             const Rect *boundsRect,
                             ConstStr255Param title,
                             Boolean visible,
                             short theProc,
                             WindowPtr behind,
                             Boolean goAwayFlag,
                             long refCon);
```

wStorage A pointer to the window record. If you specify `nil` as the value of `wStorage`, `NewWindow` allocates the window record as a nonrelocatable object in the heap. You can reduce the chances of heap fragmentation by allocating the storage from a block of memory reserved for this purpose by your application and passing a pointer to it in the `wStorage` parameter.

`boundsRect` A pointer to a rectangle, given in global coordinates, which specifies the window's initial size and location. This rectangle becomes the port rectangle of the window's graphics port. For the standard window types, `boundsRect` defines the content region of the window. The `NewWindow` function places the origin of the local coordinate system at the upper-left corner of the port rectangle.

Note

The `NewWindow` function actually calls the `QuickDraw` function `OpenPort` to create the graphics port. The bitmap, pen pattern, and other characteristics of the window's graphics port are the same as the default values set by `OpenPort`, except for the character font, which is set to the application font instead of the system font. The coordinates of the graphics port's port boundaries and visible region are changed along with its port rectangle. ♦

`title` A string that specifies the window's title.

If the title is too long to fit in the title bar, the title is truncated. If the window has a close box, characters at the end of the title are truncated; if there's no close box, the title is centered and truncated at both ends.

To suppress the title in a window with a title bar, pass an empty string, not `nil`.

`visible` A Boolean value indicating visibility status: `true` means that the Window Manager displays the window; `false` means it does not.

If the value of the `visible` parameter is `true`, the Window Manager draws a new window as soon as the window exists. The Window Manager first calls the window definition function to draw the window frame. If the value of the `goAwayFlag` parameter (described below) is also `true` and the window is frontmost (that is, if the value of the `behind` parameter is `(WindowPtr(-1L))`, the Window Manager instructs the window definition function to draw a close box in the window frame. After drawing the frame, the Window Manager generates an update event to trigger your application's drawing of the content region.

	When you create a window, you typically specify <code>false</code> as the value of the <code>visible</code> parameter. When you're ready to display the window, you call the <code>ShowWindow</code> function (page 8-513).
<code>theProc</code>	The window's definition ID, which specifies both the window definition function and the variation code for that definition function. For a list of possible values, see "Window Definition IDs" (page 8-476).
<code>behind</code>	A pointer to the window that appears immediately in front of the new window on the desktop. To place a new window in front of all other windows on the desktop, specify a value of <code>(WindowPtr)-1L</code> . When you place a new window in front of all others, <code>NewWindow</code> removes highlighting from the previously active window, highlights the newly created window, and generates activate events that trigger your application's updating of both windows. Note that if you create an invisible window in front of all others on the desktop, the user sees no active window until you make the new window visible (or make another window active). To place a new window behind all other windows, specify a value of <code>nil</code> .
<code>goAwayFlag</code>	A Boolean value that determines whether or not the window has a close box. If the value of <code>goAwayFlag</code> is <code>true</code> and the window type supports a close box, the Window Manager draws a close box in the title bar and recognizes mouse clicks in the close region; if the value of <code>goAwayFlag</code> is <code>false</code> or the window type does not support a close box, it does not.
<code>refCon</code>	The window's reference constant, set and used only by your application.

DESCRIPTION

The `NewWindow` function takes the same parameters as `NewCWindow` and, like `NewCWindow` (page 8-503), returns a `WindowPtr` as its function result. The only difference is that `NewWindow` creates a window in a monochrome graphics port, not a color graphics port. The window record and graphics port record that describe monochrome and color graphics ports are the same size and can be used interchangeably in most Window Manager functions.

The `NewWindow` function creates a window as specified by its parameters, adds it to the window list, and returns a pointer to the newly created window record. You can use the returned window pointer to refer to this window in most Window Manager functions. If `NewWindow` is unable to read the window definition function from the resource file, it returns `nil`.

If the window's definition function is not already in memory, `NewWindow` reads it into memory and stores a handle to it in the window record. It allocates space for the structure and content regions of the window and asks the window definition function to calculate those regions.

Storing the characteristics of your windows as resources, especially window titles and window items, makes your application easier to localize.

SPECIAL CONSIDERATIONS

If you let the Window Manager create the window record in your application's heap, call `DisposeWindow` (page 8-533) to close the window and dispose of its window record. If you allocated the memory for the window record yourself and passed a pointer to `NewWindow`, use the `CloseWindow` function (page 8-532) to close the window and the `DisposePtr` function (documented in *Inside Macintosh: Memory*) to dispose of the window record.

SEE ALSO

For more information about closing a window and removing the structures from memory, see the descriptions of the `DisposeWindow` function (page 8-533), the `CloseWindow` function (page 8-532), and the `DisposePtr` function (documented in *Inside Macintosh: Memory*).

Naming Windows

This section describes the functions that set and retrieve a window's title.

SetWTitle

Sets a window's title.

```
pascal void SetWTitle (WindowPtr theWindow,  
                      ConstStr255Param title);
```

theWindow A pointer to the window's window record.

title The new window title.

DESCRIPTION

The `SetWTitle` function changes a window's title to the specified string, both in the window record and on the screen, and redraws the window's frame as necessary.

When the user opens a previously saved document, you typically create a new (invisible) window with the title “untitled” and then call `SetWTitle` to give the window the document's name before displaying it. You also call `SetWTitle` when the user saves a document under a new name.

To suppress the title in a window with a title bar, pass an empty string, not `nil`.

Always use `SetWTitle` instead of directly changing the title in a window's window record.

GetWTitle

Gets a window's title.

```
pascal void GetWTitle (WindowPtr theWindow,  
                      Str255 title);
```

theWindow A pointer to the window record.

title The window title.

DESCRIPTION

The `GetWTitle` function returns the title of the window in the `title` parameter.

Your application seldom needs to determine a window's title. It might need to do so, however, when presenting user dialog boxes during operations that can affect multiple files. A spell-checking command, for example, might display a dialog box that lets the user select from all currently open documents.

When you need to retrieve a window's title, you should always use `GetWTitle` instead of reading the title from a window's window record.

Displaying Windows

This section describes the Window Manager functions that change a window's display and position in the window list but not its size or location on the desktop. Note that the Window Manager automatically draws all visible windows on the screen.

Your application typically uses only a few of the functions described in this section: `DrawGrowIcon` (page 8-511), `SelectWindow` (page 8-512), `ShowWindow` (page 8-513), and, occasionally, `HideWindow` (page 8-513).

DrawGrowIcon

Draws a window's size box.

```
pascal void DrawGrowIcon (WindowPtr theWindow);
```

`theWindow` A pointer to the window's window record.

DESCRIPTION

The `DrawGrowIcon` function draws a window's size box or, if the window can't be sized, whatever other image is appropriate. You call `DrawGrowIcon` when drawing the content region of a window that contains a size box.

The exact appearance and location of the image depend on the window type and the window's active or inactive state. The `DrawGrowIcon` function automatically checks the window's type and state and draws the appropriate image.

In an active document window, `DrawGrowIcon` draws the grow image in the size box in the lower-right corner of the window's graphics port rectangle, along with the lines delimiting the size box and scroll bar areas. To draw the size box but not the scroll bar outline, set the `clipRgn` field in the window's graphics port to be a 15-by-15 pixel rectangle in the lower-right corner of the window.

The `DrawGrowIcon` function doesn't erase the scroll bar areas. If you use `DrawGrowIcon` to draw the size box and scroll bar outline, therefore, you should erase those areas yourself when the window size changes, even if the window doesn't contain scroll bars.

In an inactive document window, `DrawGrowIcon` draws the lines delimiting the size box and scroll bar areas and erases the size box.

SelectWindow

Makes a window active.

```
pascal void SelectWindow (WindowPtr theWindow);
```

theWindow A pointer to the window's window record.

DESCRIPTION

The `SelectWindow` function removes highlighting from the previously active window, brings the specified window to the front, highlights it, and generates the activate events to deactivate the previously active window and activate the specified window. If the specified window is already active, `SelectWindow` has no effect.

Even if the specified window is invisible, `SelectWindow` brings the window to the front, activates the window, and deactivates the previously active window. Note that in this case, no active window is visible on the screen. If you do select an invisible window, be sure to call `ShowWindow` (page 8-513) immediately to make the window visible (and accessible to the user).

Call `SelectWindow` when the user presses the mouse button while the cursor is in the content region of an inactive window.

ShowWindow

Makes an invisible window visible.

```
pascal void ShowWindow (WindowPtr theWindow);
```

theWindow A pointer to the window record of the window.

DESCRIPTION

The `ShowWindow` function makes an invisible window visible. If the specified window is already visible, `ShowWindow` has no effect. Your application typically creates a new window in an invisible state, performs any necessary setup of the content region, and then calls `ShowWindow` to make the window visible.

When you display a previously invisible window by calling `ShowWindow`, the Window Manager draws the window frame and then generates an update event to trigger your application's drawing of the content region.

If the newly visible window is the frontmost window, `ShowWindow` highlights it if it's not already highlighted and generates an activate event to make it active. The `ShowWindow` function does not activate a window that is not frontmost on the desktop.

Note

Because `ShowWindow` does not change the front-to-back ordering of windows, it is not the inverse of `HideWindow` (page 8-513). If you make the frontmost window invisible with `HideWindow`, and `HideWindow` has activated another window, you must call both `ShowWindow` and `SelectWindow` (page 8-512) to bring the original window back to the front. ♦

HideWindow

Makes a window invisible.

```
pascal void HideWindow (WindowPtr theWindow);
```

`theWindow` A pointer to the window's window record.

DESCRIPTION

The `HideWindow` function make a visible window invisible. If you hide the frontmost window, `HideWindow` removes the highlighting, brings the window behind it to the front, highlights the new frontmost window, and generates the appropriate activate events.

To reverse the actions of `HideWindow`, you must call both `ShowWindow` (page 8-513), to make the window visible, and `SelectWindow` (page 8-512), to select it.

ShowHide

Sets a window's visibility.

```
pascal void ShowHide (WindowPtr theWindow,
                      Boolean showFlag);
```

`theWindow` A pointer to the window's window record.

`showFlag` A Boolean value that determines visibility status: `true` makes a window visible; `false` makes it invisible.

DESCRIPTION

The `ShowHide` function sets a window's visibility to the status specified by the `showFlag` parameter. If the value of `showFlag` is `true`, `ShowHide` makes the window visible if it's not already visible and has no effect if it's already visible. If the value of `showFlag` is `false`, `ShowHide` makes the window invisible if it's not already invisible and has no effect if it's already invisible.

The `ShowHide` function never changes the highlighting or front-to-back ordering of windows and generates no activate events.

▲ **WARNING**

Use this function carefully and only in special circumstances where you need more control than that provided by `HideWindow` (page 8-513) and `ShowWindow` (page 8-513). Do not, for example, use `ShowHide` to hide the active window without making another window active. ▲

HiliteWindow

Sets a window's highlighting status.

```
pascal void HiliteWindow (WindowPtr theWindow,
                          Boolean fHilite);
```

<code>theWindow</code>	A pointer to the window's window record.
<code>fHilite</code>	A Boolean value that determines the highlighting status: <code>true</code> highlights a window; <code>false</code> removes highlighting.

DESCRIPTION

The `HiliteWindow` function sets a window's highlighting status to the specified state. If the value of the `fHilite` parameter is `true`, `HiliteWindow` highlights the specified window; if the specified window is already highlighted, the function has no effect. If the value of `fHilite` is `false`, `HiliteWindow` removes highlighting from the specified window; if the window is not already highlighted, the function has no effect.

Your application doesn't normally need to call `HiliteWindow`. To make a window active, you can call `SelectWindow` (page 8-512), which handles highlighting for you.

BringToFront

Brings a window to the front.

```
pascal void BringToFront (WindowPtr theWindow);
```

`theWindow` A pointer to the window's window record.

DESCRIPTION

The `BringToFront` function puts the specified window at the beginning of the window list and redraws the window in front of all others on the screen. It does not change the window's highlighting or make it active.

Your application does not ordinarily call `BringToFront`. The user interface guidelines specify that the frontmost window should be the active window. To bring a window to the front and make it active, call the `SelectWindow` function (page 8-512).

SendBehind

Moves one window behind another.

```
pascal void SendBehind (WindowPtr theWindow,
                        WindowPtr behindWindow);
```

`theWindow` A pointer to the window to be moved.

`behindWindow` A pointer to the window that is to be in front of the moved window.

DESCRIPTION

The `SendBehind` function moves the window pointed to by the parameter `theWindow` behind the window pointed to by the parameter `behindWindow`. If the move exposes previously obscured windows or parts of windows, `SendBehind` redraws the frames as necessary and generates the appropriate update events to have any newly exposed content areas redrawn.

If the value of `behindWindow` is `nil`, `SendBehind` sends the window to be moved behind all other windows on the desktop. If the window to be moved is the active window, `SendBehind` removes its highlighting, highlights the newly exposed frontmost window, and generates the appropriate activate events.

Note

Do not use `SendBehind` to deactivate a window after you've made a new window active with the `SelectWindow` function (page 8-512). The `SelectWindow` function automatically deactivates the previously active window. ♦

Retrieving Window Information

This section describes

- the `FindWindow` function (page 8-517), which maps the cursor location of a mouse-down event to parts of the screen or regions of a window
- the `FrontWindow` function (page 8-518), which tells your application which window is active

FindWindow

Maps the location of the cursor to a part of the screen or a region of a window when your application receives a mouse-down event.

```
pascal short FindWindow (Point thePoint,
                        WindowPtr *theWindow);
```

`thePoint` The point, in global coordinates, where the mouse-down event occurred. Your application retrieves this information from the `where` field of the event record.

`theWindow` A pointer to the address where `FindWindow` returns a pointer to the window in which the mouse-down event occurred, if it occurred in a window. If it didn't occur in a window, `FindWindow` sets the value of the address pointed to by `theWindow` to `nil`.

DESCRIPTION

The `FindWindow` function returns an integer that specifies where the cursor was when the user pressed the mouse button. You typically call `FindWindow` whenever you receive a mouse-down event. The `FindWindow` function helps you dispatch the event by reporting whether the cursor was in the menu bar or in a

window when the mouse button was pressed and, if it was in a window, which window and which region of the window. If the mouse-down event occurred in a window, `FindWindow` places a pointer to the window in the value referenced through the parameter `theWindow`.

The `FindWindow` function returns an integer that specifies one of nine regions. For a list of the corresponding enumerators and the actions your application should take, see “FindWindow Result Codes” (page 8-479).

FrontWindow

Identifies the active window.

```
pascal WindowPtr FrontWindow (void);
```

DESCRIPTION

The `FrontWindow` function returns a pointer to the first visible window in the window list (that is, the active window). If there are no visible windows, `FrontWindow` returns `nil`.

SPECIAL CONSIDERATIONS

The `FrontWindow` function may move memory. Therefore, you should not use it in a VBL task.

Moving Windows

This section describes the functions that move windows on the desktop.

To move a window, your application ordinarily needs to call only the `DragWindow` function (page 8-519), which itself calls the `DragGrayRgn` function (page 8-521), and the `MoveWindow` function (page 8-520). The `DragGrayRgn` function drags a dotted outline of the window on the screen, following the motion of the cursor, as long as the user holds down the mouse button. The `DragGrayRgn` function itself calls the `PinRect` function (page 8-525) to contain the point where the cursor was when the mouse button was first pressed inside the

available desktop area. When the user releases the mouse button, `DragWindow` calls `MoveWindow`, which moves the window to a new location.

DragWindow

Moves a window on the screen when the user drags it by its title bar.

```
pascal void DragWindow (WindowPtr theWindow,
                        Point startPt,
                        const Rect *boundsRect);
```

<code>theWindow</code>	A pointer to the window record of the window to be dragged.
<code>startPt</code>	The location, in global coordinates, of the cursor at the time the user pressed the mouse button. Your application retrieves this point from the <code>where</code> field of the event record.
<code>boundsRect</code>	<p>A pointer to a rectangle, given in global coordinates, that limits the region to which a window can be dragged. If the mouse button is released when the cursor is outside the limits of <code>boundsRect</code>, <code>DragWindow</code> returns without moving the window (or, if it was inactive, without making it the active window).</p> <p>Because the user cannot ordinarily move the cursor off the desktop, you can safely set <code>boundsRect</code> to the largest available rectangle (the bounding box of the desktop region pointed to by the global variable <code>GrayRgn</code>) when you're using <code>DragWindow</code> to track mouse movements. Don't set the bounding rectangle to the size of the immediate screen (<code>screenBits.bounds</code>), because the user wouldn't be able to move the window to a different screen on a system equipped with multiple monitors.</p>

DESCRIPTION

The `DragWindow` function uses `DragGrayRgn` (page 8-521) to move a dotted outline of the specified window around the screen, following the movement of the cursor until the user releases the mouse button. When the button is released, `DragWindow` calls `MoveWindow` (page 8-520) to move the window to its new location. If the specified window isn't the active window (and the Command key wasn't down when the mouse button was pressed), `DragWindow` makes it the

active window by setting the `front` parameter to `true` when calling `MoveWindow`. If the Command key was down when the mouse button was pressed, `DragWindow` moves the window without making it active.

MoveWindow

Moves a window on the desktop.

```
pascal void MoveWindow (WindowPtr theWindow,
                        short hGlobal,
                        short vGlobal,
                        Boolean front);
```

<code>theWindow</code>	A pointer to the window record of the window being moved.
<code>hGlobal</code>	The new location, in global coordinates, of the left edge of the window's port rectangle.
<code>vGlobal</code>	The new location, in global coordinates, of the top edge of the window's port rectangle.
<code>front</code>	A Boolean value specifying whether the window is to become the frontmost, active window. If the value of the <code>front</code> parameter is <code>false</code> , <code>MoveWindow</code> does not change its plane or status. If the value of the <code>front</code> parameter is <code>true</code> and the window isn't active, <code>MoveWindow</code> makes it active by calling the <code>SelectWindow</code> function.

DESCRIPTION

The `MoveWindow` function moves the specified window to the location specified by the `hGlobal` and `vGlobal` parameters, without changing the window's size. The upper-left corner of the window's port rectangle is placed at the point (`vGlobal`, `hGlobal`). The local coordinates of the upper-left corner are unaffected.

Your application doesn't normally call `MoveWindow`. When the user drags a window by dragging its title bar, you can call `DragWindow` (page 8-519) which in turn calls `MoveWindow` when the user releases the mouse button.

DragGrayRgn

The `DragWindow` function (page 8-519) calls the `DragGrayRgn` function to move an outline of a window around the screen as the user drags a window.

```
pascal long DragGrayRgn (RgnHandle theRgn,
                        Point startPt,
                        const Rect *limitRect,
                        const Rect *slopRect,
                        short axis,
                        DragGrayRgnProcUPP actionProc);
```

`theRgn` A handle to the region to be dragged.

`startPt` The location, in the local coordinates of the current graphics port, of the cursor when the mouse button was pressed.

`limitRect` A pointer to a rectangle, given in the local coordinates of the current graphics port, that limits where the region can be dragged. This parameter works with the `slopRect` parameter, as illustrated in Figure 7-2 (page 8-524).

`slopRect` A pointer to a rectangle, given in the local coordinates of the current graphics port, that gives the user some leeway in moving the mouse without violating the limits of the `limitRect` parameter, as illustrated in Figure 7-2 (page 8-524). The `slopRect` rectangle should be larger than the `limitRect` rectangle.

`axis` A constant that constrains the region's motion. The `axis` parameter can have one of these values (defined by the Control Manager):

```
enum {
    noConstraint      = 0,      /* no constraints */
    hAxisOnly         = 1,      /* move on horizontal axis
                                /* only */
    vAxisOnly         = 2      /* move on vertical axis */
                                /* only */
};
```

If an axis constraint is in effect, the outline follows the cursor's movements along only the specified axis, ignoring motion along the other axis. With or without an axis constraint, the outline appears only when the mouse is inside the `slopRect` rectangle.

`actionProc`

A pointer to a function that defines an action to be performed repeatedly as long as the user holds down the mouse button. The function can have no parameters. If the value of `actionProc` is `nil`, `DragGrayRgn` simply retains control until the mouse button is released.

DESCRIPTION

The `DragGrayRgn` function moves a gray outline of a region on the screen, following the movements of the cursor, until the mouse button is released. It returns the difference between the point where the mouse button was pressed and the **offset point**—that is, the point in the region whose horizontal and vertical offsets from the upper-left corner of the region's enclosing rectangle are the same as the offsets of the starting point when the user pressed the mouse button. The `DragGrayRgn` function stores the vertical difference between the starting point and the offset point in the high-order word of the return value and the horizontal difference in the low-order word.

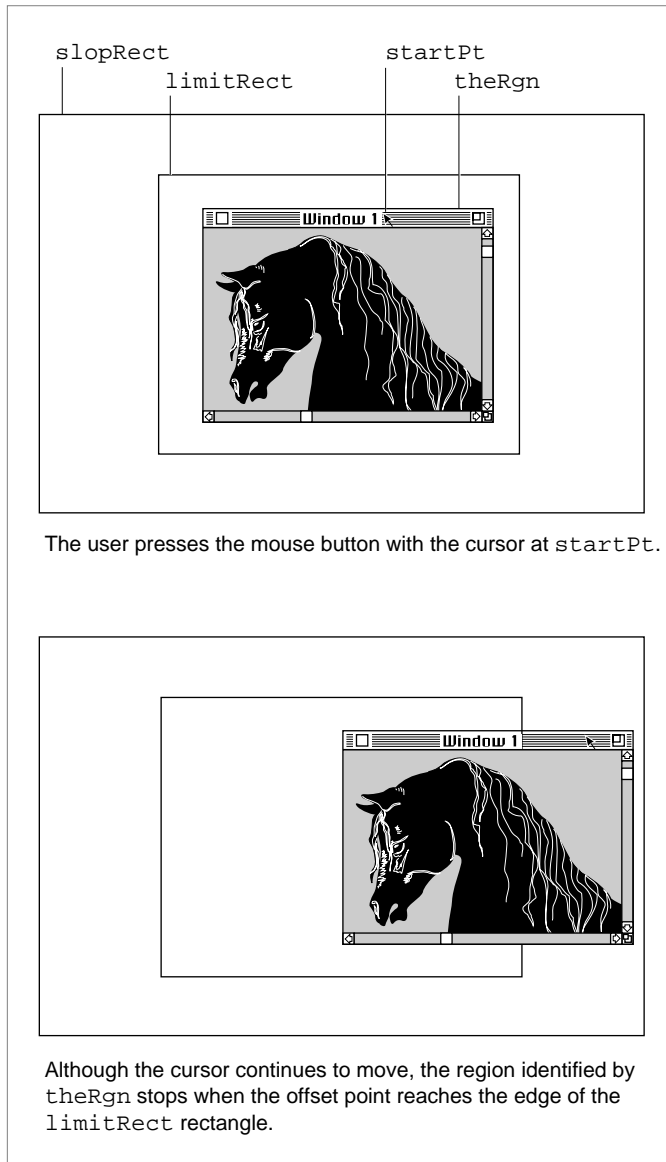
The `DragGrayRgn` function limits the movement of the region according to the constraints set by the `limitRect` and `slopRect` parameters:

- As long as the cursor is inside the `limitRect` rectangle, the region's outline follows it normally. If the mouse button is released while the cursor is within this rectangle, the return value reflects the simple distance that the cursor moved in each dimension.
- When the cursor moves outside the `limitRect` rectangle, the offset point stops at the edge of the `limitRect` rectangle. If the mouse button is released while the cursor is outside the `limitRect` rectangle but inside the `slopRect` rectangle, the return value reflects only the difference between the starting point and the offset point, regardless of how far outside of the `limitRect` rectangle the cursor may have moved. (Note that part of the region can fall outside the `limitRect` rectangle, but not the offset point.)
- When the cursor moves outside the `slopRect` rectangle, the region's outline disappears from the screen. The `DragGrayRgn` function continues to track the cursor, however, and if the cursor moves back into the `slopRect` rectangle, the outline reappears. If the mouse button is released while the cursor is outside

the `slopRect` rectangle, both words of the return value are set to \$8000. In this case, the Window Manager does not move the window from its original location.

Figure 7-2 (page 8-524) illustrates how the region stops moving when the offset point reaches the edge of the `limitRect` rectangle. The cursor continues to move, but the region does not.

If the mouse button is released while the cursor is anywhere inside the `slopRect` rectangle, the Window Manager redraws the window in its new location, which is calculated from the value returned by `DragGrayRgn`.

Figure 7-2 Limiting rectangle used by DragGrayRgn

ASSEMBLY-LANGUAGE INFORMATION

You can set the global variable `DragHook` to point to an optional function, defined by your application, which will be called by `DragGrayRgn` as long as the mouse button is held down. (If there's an `actionProc` function, it is called first.) If you want `DragGrayRgn` to draw the region's outline in a pattern other than gray, you can store the pattern in the global variable `DragPattern` and then invoke the macro `_DragTheRgn`. Note that the use of the Window Manager's global variables is not guaranteed to be compatible with system software versions later than System 6.

PinRect

The `DragGrayRgn` function uses the `PinRect` function to contain a point within a specified rectangle.

```
pascal long PinRect (const Rect *theRect, Point *thePt);
```

<code>theRect</code>	A pointer to a rectangle in which the point is to be contained.
<code>thePt</code>	A pointer to the point to be contained.

DESCRIPTION

The `PinRect` function returns a point within the specified rectangle that is as close as possible to the specified point. (The high-order word of the returned long integer is the vertical coordinate; the low-order word is the horizontal coordinate.)

If the specified point is within the rectangle, `PinRect` returns the point itself. If not, then

- if the horizontal position is to the left of the rectangle, `PinRect` returns the left edge as the horizontal coordinate
- if the horizontal position is to the right of the rectangle, `PinRect` returns the right edge minus 1 as the horizontal coordinate
- if the vertical position is above the rectangle, `PinRect` returns the top edge as the vertical coordinate

- if the vertical position is below the rectangle, `PinRect` returns the bottom edge minus 1 as the vertical coordinate

Note

The 1 is subtracted when the point is below or to the right of the rectangle so that a pixel drawn at that point lies within the rectangle. If the point is exactly on the bottom or the right edge of the rectangle, however, 1 should be subtracted but isn't. ♦

Resizing Windows

This section describes the functions you can use to track the cursor while the user resizes a window and to draw the window in a new size.

GrowWindow

Allows the user to change the size of a window.

```
pascal long GrowWindow (WindowPtr theWindow,
                        Point startPt,
                        const Rect *bBox);
```

<code>theWindow</code>	A pointer to the window record of the window to drag.
<code>startPt</code>	The location of the cursor at the time the mouse button was first pressed, in global coordinates. Your application retrieves this point from the <code>where</code> field of the event record.
<code>sizeRect</code>	<p>A pointer to a rectangle structure that specifies the limits on the vertical and horizontal measurements of the port rectangle, in pixels.</p> <p>Although the <code>sizeRect</code> parameter gives the address of a structure which is in the form of the <code>Rect</code> data type, the four numbers in the structure represent lengths, not screen coordinates. The <code>top</code>, <code>left</code>, <code>bottom</code>, and <code>right</code> fields of the <code>sizeRect</code> parameter specify the minimum vertical measurement</p>

(top), the minimum horizontal measurement (left), the maximum vertical measurement (bottom), and the maximum horizontal measurement (right).

The minimum measurements must be large enough to allow a manageable rectangle; 64 pixels on a side is typical. Because the user cannot ordinarily move the cursor off the screen, you can safely set the upper bounds to the largest possible length (65,535 pixels) when you're using `GrowWindow` to follow cursor movements.

DESCRIPTION

The `GrowWindow` function displays an outline (grow image) of the window as the user moves the cursor to make the window larger or smaller; it handles all user interaction until the user releases the mouse button. After calling `GrowWindow`, you call the `SizeWindow` function (page 8-528) to change the size of the window.

The `GrowWindow` function moves a dotted-line image of the window's right and lower edges around the screen, following the movements of the cursor until the mouse button is released. It returns the new dimensions, in pixels, of the resulting window: the height in the high-order word of the returned long-integer value and the width in the low-order word. You can use the functions `HiWord` and `LoWord` (described in *Inside Macintosh: Operating System Utilities*) to retrieve only the high-order and low-order words, respectively.

A return value of 0 means that the new size is the same as the size of the current port rectangle.

ASSEMBLY-LANGUAGE INFORMATION

You can set the global variable `DragHook` to point to an optional function, defined by your application, which will be called by `GrowWindow` as long as the mouse button is held down. (If there's an `actionProc` function, the `actionProc` function is called first.) Note that the use of the Window Manager's global variables is not guaranteed to be compatible with system software versions later than System 6.

SizeWindow

Sets the size of a window.

```
pascal void SizeWindow (WindowPtr theWindow,
                        short w,
                        short h,
                        Boolean fUpdate);
```

<code>theWindow</code>	A pointer to the window record of the window to be sized.
<code>w</code>	The new window width, in pixels.
<code>h</code>	The new window height, in pixels.
<code>fUpdate</code>	A Boolean value that specifies whether any newly created area of the content region is to be accumulated into the update region (<code>true</code>) or not (<code>false</code>). You ordinarily pass a value of <code>true</code> to ensure that the area is updated. If you pass <code>false</code> , you're responsible for maintaining the update region yourself. For more information on adding rectangles to and removing rectangles from the update region, see <code>InvalidRect</code> (page 8-535) and <code>ValidRect</code> (page 8-536).

DESCRIPTION

The `SizeWindow` function changes the size of the window's graphics port rectangle to the dimensions specified by the `w` and `h` parameters, or does nothing if the values of `w` and `h` are 0. The Window Manager redraws the window in the new size, recentering the title and truncating it if necessary. Your application calls `SizeWindow` immediately after calling `GrowWindow` (page 8-526) to adjust the window to any changes made by the user through the size box.

Zooming Windows

This section describes the functions you can use to track mouse activity in the zoom box and to zoom windows.

TrackBox

Tracks the cursor when the user presses the mouse button while the cursor is in the zoom box.

```
pascal Boolean TrackBox (WindowPtr theWindow,
                        Point thePt,
                        short partCode);
```

theWindow	A pointer to the window record of the window in which the mouse button was pressed.
thePt	The location of the cursor when the mouse button was pressed. Your application receives this point from the <code>where</code> field in the event record.
partCode	The part code (either <code>inZoomIn</code> or <code>inZoomOut</code>) returned by the <code>FindWindow</code> function (page 8-517).

DESCRIPTION

The `TrackBox` function tracks the cursor when the user presses the mouse button while the cursor is in the zoom box, retaining control until the mouse button is released. While the button is down, `TrackBox` highlights the zoom box while the cursor is in the zoom region.

When the mouse button is released, `TrackBox` removes the highlighting from the zoom box and returns `true` if the cursor is within the zoom region and `false` if it is not.

Your application calls the `TrackBox` function when it receives a result code of either `inZoomIn` or `inZoomOut` from the `FindWindow` function (page 8-517). If `TrackBox` returns `true`, your application calculates the standard state, if necessary, and calls the `ZoomWindow` function (page 8-530) to zoom the window. If `TrackBox` returns `false`, your application does nothing.

ASSEMBLY-LANGUAGE INFORMATION

You can set the global variable `DragHook` to point to an optional function, defined by your application, which will be called by `TrackBox` as long as the mouse button is held down. (If there's an `actionProc` function, the `actionProc` function is called first.) Note that the use of the Window Manager's global

variables is not guaranteed to be compatible with system software versions later than System 6.

ZoomWindow

Zooms the window when the user has pressed and released the mouse button with the cursor in the zoom box.

```
pascal void ZoomWindow (WindowPtr theWindow,
                        short partCode,
                        Boolean front);
```

<code>theWindow</code>	A pointer to the window record for the window to be zoomed.
<code>partCode</code>	The result (either <code>inZoomIn</code> or <code>inZoomOut</code>) returned by the <code>FindWindow</code> function (page 8-517).
<code>front</code>	A Boolean value that determines whether the window is to be brought to the front. If the value of <code>front</code> is <code>true</code> , the window necessarily becomes the frontmost, active window. If the value of <code>front</code> is <code>false</code> , the window's position in the window list does not change. Note that if a window was active before it was zoomed, it remains active even if the value of <code>front</code> is <code>false</code> .

DESCRIPTION

The `ZoomWindow` function zooms a window in or out, depending on the value of the `partCode` parameter. Your application calls `ZoomWindow`, passing it the part code returned by `FindWindow` (page 8-517), when it receives a result of `true` from `TrackBox`. The `ZoomWindow` function then changes the window's port rectangle to either the user state (if the part code is `inZoomIn`) or the standard state (if the part code is `inZoomOut`), as stored in the window state data record.

If the part code is `inZoomOut`, your application ordinarily calculates and sets the standard state before calling `ZoomWindow`.

For best results, call the QuickDraw function `EraseRect`, passing the window's graphics port as the port rectangle, before calling `ZoomWindow`.

Closing and Deallocating Windows

This section describes the functions that track user activity in the close box and that close and dispose of windows.

When you no longer need a window, call the `CloseWindow` function (page 8-532) if you allocated the memory for the window record or the `DisposeWindow` function (page 8-533) if you did not.

TrackGoAway

Tracks the cursor when the user presses the mouse button while the cursor is in the close box.

```
pascal Boolean TrackGoAway (WindowPtr theWindow,
                             Point thePt);
```

`theWindow` A pointer to the window record of the window in which the mouse-down event occurred.

`thePt` The location of the cursor at the time the mouse button was pressed. Your application receives this point from the `where` field of the event record.

DESCRIPTION

The `TrackGoAway` function tracks cursor activity when the user presses the mouse button while the cursor is in the close box, retaining control until the user releases the mouse button. While the button is down, `TrackGoAway` highlights the close box as long as the cursor is in the close region.

When the mouse button is released, `TrackGoAway` removes the highlighting from the close box and returns `true` if the cursor is within the close region and `false` if it is not.

Your application calls the `TrackGoAway` function when it receives a result code of `inGoAway` from the `FindWindow` function (page 8-517). If `TrackGoAway` returns `true`, your application calls its own function for closing a window, which can call either the `CloseWindow` function (page 8-532) or the `DisposeWindow` function (page 8-533) to remove the window from the screen. (Before removing a document window, your application ordinarily checks whether the document

has changed since the associated file was last saved. See the chapter “Introduction to File Management” in *Inside Macintosh: Files* for a general discussion of handling files.) If `TrackGoAway` returns `false`, your application does nothing.

ASSEMBLY-LANGUAGE INFORMATION

You can set the global variable `DragHook` to point to an optional function, defined by your application, which will be called by `TrackGoAway` as long as the mouse button is held down. (If there’s an `actionProc` function, the `actionProc` function is called first.) Note that the use of the Window Manager’s global variables is not guaranteed to be compatible with system software versions later than System 6.

CloseWindow

Removes a window if you allocated memory yourself for the window’s window record.

```
pascal void CloseWindow (WindowPtr theWindow);
```

`theWindow` A pointer to the window record for the window to be closed.

DESCRIPTION

The `CloseWindow` function removes the specified window from the screen and deletes it from the window list. It releases the memory occupied by all data structures associated with the window except the window record itself.

If you allocated memory for the window record and passed a pointer to it as one of the parameters to the functions that create windows, call `CloseWindow` when you’re done with the window. You must then call the Memory Manager routine `DisposePtr` to release the memory occupied by the window record.

▲ **WARNING**

If your application allocated any other memory for use with a window, you must release it before calling `CloseWindow`. The Window Manager releases only the data structures it created.

Also, `CloseWindow` assumes that any picture pointed to by the window record field `windowPic` is data, not a resource, and it calls the QuickDraw function `KillPicture` to delete it. If your application uses a picture stored as a resource, you must release the memory it occupies with the `ReleaseResource` function and set the `windowPic` field to `nil` before closing the window. ▲

Any pending update events for the window are discarded. If the window being removed is the frontmost window, the window behind it, if any, becomes the active window.

DisposeWindow

Removes a window if you let the Window Manager allocate memory for the window record.

```
pascal void DisposeWindow (WindowPtr theWindow);
```

`theWindow` A pointer to the window record of the window to be closed.

DESCRIPTION

The `DisposeWindow` function calls `CloseWindow` (page 8-532) to remove a window from the screen and deletes it from the window list, then releases the memory occupied by all structures associated with the window, including the window record.

▲ WARNING

If your application allocated any other memory for use with a window, you must release it before calling `DisposeWindow`. The Window Manager releases only the data structures it created.

The `DisposeWindow` function assumes that any picture pointed to by the window record field `windowPic` is data, not a resource, and it calls the `QuickDraw` function `KillPicture` to delete it. If your application uses a picture stored as a resource, you must release the memory it occupies with the `ReleaseResource` function and set the `windowPic` field to `nil` before closing the window. ▲

Any pending update events for the window are discarded. If the window being removed is the frontmost window, the window behind it, if any, becomes the active window.

Maintaining the Update Region

This section describes the functions you use to update your windows and to maintain window update regions.

BeginUpdate

Starts updating a window when you receive an update event for that window.

```
pascal void BeginUpdate (WindowPtr theWindow);
```

`theWindow` A pointer to the window's window record. Your application gets this information from the `message` field in the update event record.

DESCRIPTION

The `BeginUpdate` function limits the visible region of the window's graphics port to the intersection of the visible region and the update region; it then sets the window's update region to an empty region. After calling `BeginUpdate`, your application redraws either the entire content region or only the visible region. In

either case, only the parts of the window that require updating are actually redrawn on the screen.

Every call to `BeginUpdate` must be matched with a subsequent call to `EndUpdate` (page 8-535) after your application redraws the content region. `BeginUpdate` and `EndUpdate` can't be nested. That is, you must call `EndUpdate` before the next call to `BeginUpdate`.

SPECIAL CONSIDERATIONS

If you don't clear the update region when you receive an update event, the Event Manager continues to send update events until you do.

EndUpdate

Finishes updating a window.

```
pascal void EndUpdate (WindowPtr theWindow);
```

`theWindow` A pointer to the window's window record.

DESCRIPTION

The `EndUpdate` function restores the normal visible region of a window's graphics port. When you receive an update event for a window, you call `BeginUpdate` (page 8-534), redraw the update region, and then call `EndUpdate`. Each call to `BeginUpdate` must be balanced by a subsequent call to `EndUpdate`.

InvalRect

Adds a rectangle to a window's update region.

```
pascal void InvalRect (const Rect *badRect);
```

`badRect` A pointer to a rectangle, in local coordinates, that is to be added to a window's update region.

DESCRIPTION

The `InvalidRect` function adds a specified rectangle to the update region of the window whose graphics port is the current port. Specify the rectangle in local coordinates. The Window Manager clips it, if necessary, to fit in the window's content region.

Both your application and the Window Manager use the `InvalidRect` function. When the user enlarges a window, for example, the Window Manager uses `InvalidRect` to add the newly created content region to the update region. Your application uses `InvalidRect` to add the two rectangles formerly occupied by the scroll bars in the smaller content area.

InvalidRgn

Adds a region to a window's update region.

```
pascal void InvalidRgn (RgnHandle badRgn);
```

`badRgn` The region, in local coordinates, that is to be added to a window's update region.

DESCRIPTION

The `InvalidRgn` function adds a specified region to the update region of the window whose graphics port is the current port. Specify the region in local coordinates. The Window Manager clips it, if necessary, to fit in the window's content region.

ValidRect

Removes a rectangle from a window's update region.

```
pascal void ValidRect (const Rect *goodRect);
```

`goodRect` A pointer to a rectangle, in local coordinates, to be removed from a window's update region.

DESCRIPTION

The `ValidRect` function removes a specified rectangle from the update region of the window whose graphics port is the current port. Specify the rectangle in local coordinates. The Window Manager clips it, if necessary, to fit in the window's content region.

Your application uses `ValidRect` to tell the Window Manager that it has already drawn a rectangle and to cancel any updates accumulated for that area. You can thereby improve response time by reducing redundant redrawing.

Suppose, for example, that you've resized a window that contains a size box and scroll bars. Depending on the dimensions of the newly sized window, the new size box and scroll bar areas may or may not have been accumulated into the window's update region. After calling `SizeWindow` (page 8-528), you can redraw the size box or scroll bars immediately and then call `ValidRect` for the areas they occupy. If they were in fact accumulated into the update region, `ValidRect` removes them so that you do not have to redraw them with the next update event.

ValidRgn

Removes a specified region from a window's update region.

```
pascal void ValidRgn (RgnHandle goodRgn);
```

`goodRgn` A region, in local coordinates, to be removed from a window's update region.

DESCRIPTION

The `ValidRgn` function removes a specified region from the update region of the window whose graphics port is the current port. Specify the region in local coordinates. The Window Manager clips it, if necessary, to fit in the window's content region.

Setting and Retrieving Other Window Characteristics

This section describes the functions that let you set and retrieve less commonly used fields in the window record.

SetWindowPic

Sets a picture for the Window Manager to draw in a window's content region.

```
pascal void SetWindowPic (WindowPtr theWindow,  
                          PicHandle pic);
```

theWindow	A pointer to a window's window record.
Pic	A handle to the picture to be drawn in the window.

DESCRIPTION

The `SetWindowPic` function stores in a window's window record a handle to a picture to be drawn in the window. When the window's content region must be updated, the Window Manager then draws the picture or part of the picture, as necessary, instead of generating an update event.

Note

The `CloseWindow` (page 8-532) and `DisposeWindow` (page 8-533) functions assume that any picture pointed to by the window record field `windowPic` is stored as data, not as a resource. If your application uses a picture stored as a resource, you must release the memory it occupies by calling the Resource Manager's `ReleaseResource` function and set the `WindowPic` field to `nil` before you close the window. ♦

GetWindowPic

Returns a handle to a window's picture.

```
pascal PicHandle GetWindowPic (WindowPtr theWindow);
```

theWindow	A pointer to the window's window record.
-----------	--

DESCRIPTION

The `GetWindowPic` function returns a handle to the picture to be drawn in a specified window's content region. The handle must have been stored previously with the `SetWindowPic` function (page 8-538).

SetWRefCon

Sets the `refCon` field of a window record.

```
pascal void SetWRefCon (WindowPtr theWindow,  
                        long data);
```

`theWindow` A pointer to the window's window record.

`data` The data to be placed in the `refCon` field.

DESCRIPTION

The `SetWRefCon` function places the specified data in the `refCon` field of the specified window record. The `refCon` field is available to your application for any window-related data it needs to store.

GetWRefCon

Returns the reference constant from a window's window record.

```
pascal long GetWRefCon (WindowPtr theWindow);
```

`theWindow` A pointer to the window's window record.

DESCRIPTION

The `GetWRefCon` function returns the long integer data stored in the `refCon` field of the specified window record.

GetWVariant

Returns a window's variation code.

```
pascal short GetWVariant (WindowPtr theWindow);
```

`theWindow` A pointer to the window's window record.

DESCRIPTION

The `GetWVariant` function returns the variation code of the specified window. Depending on the window's window definition function, the result of `GetWVariant` can represent one of the standard window types.

Manipulating the Desktop

This section describes the functions that let your application retrieve information about the desktop and set the desktop pattern. Ordinarily, your application doesn't need to manipulate any part of the desktop outside of its own windows.

SetDeskCPat

Sets the desktop pattern on a computer that supports Color QuickDraw.

```
pascal void SetDeskCPat (PixPatHandle deskPixPat);
```

`deskPixPat` A handle to a pixel pattern.

DESCRIPTION

The `SetDeskCPat` function sets the desktop pattern to a specified pixel pattern, which can be drawn in more than two colors. After a call to `SetDeskCPat`, the desktop is automatically redrawn in the new pattern. If the specified pattern is a binary pattern (with a pattern type of 0), it is drawn in the current foreground and background colors. If the value of the `deskPixPat` parameter is `nil`,

`SetDeskCPat` uses the standard binary desk pattern (that is, the 'ppat' resource with resource ID 16).

Note

For compatibility with other Macintosh applications and the system software, applications should ordinarily not change the desktop pattern. ♦

The Window Manager's desktop-painting functions can paint the desktop either in the binary pattern stored in the global variable `DeskPattern` or in a new pixel pattern. The desktop pattern used at startup is determined by the value of the parameter-RAM bit flag called `pCDeskPat`. If the value of `pCDeskPat` is 0, the Window Manager uses the new pixel pattern; if not, it uses the binary pattern stored in `DeskPattern`. The user can change the color pattern through the General Controls panel, which changes the value of `pCDeskPat`.

LMGetDeskPattern

Gets the value of the global variable `DeskPattern`.

```
pascal void LMGetDeskPattern (Pattern *deskPatternValue);
```

`deskPatternValue`

A pointer to the address at which `LMGetDeskPattern` returns the pixel pattern contained in the global variable `DeskPattern`.

LMSetGrayRgn

Sets the handle to the region interpreted by the Window Manager as the current desktop region.

```
pascal void LMSetGrayRgn (RgnHandle value);
```

`value`

A handle to the region you want to set as the current desktop region.

DESCRIPTION

The `LMSetGrayRgn` function sets the value of the low-memory global `GrayRgn`, which contains a handle to a region interpreted by the Window Manager as the current desktop region.

SEE ALSO

To get a handle to the current desktop region, use `GetGrayRgn` (page 8-542).

GetGrayRgn

Returns a handle to the current desktop region.

```
pascal RgnHandle GetGrayRgn (void);
```

DESCRIPTION

The `GetGrayRgn` function returns a handle to the current desktop region from the global variable `GrayRgn`.

The desktop region represents all available screen space, that is, the desktop area displayed by all monitors attached to the computer. Ordinarily, your application doesn't need to access the desktop region directly.

When your application calls `DragWindow` (page 8-519) to let the user drag a window, it can use `GetGrayRgn` to set the limiting rectangle to the entire desktop area.

LMSetWMgrPort

Sets the Window Manager port.

```
pascal void LMSetWMgrPort (GrafPtr value);
```

<code>value</code>	A pointer to the graphics port you want to set as the Window Manager port.
--------------------	--

DESCRIPTION

The `LMSetGrayRgn` function sets the value of the low-memory global `WMgrPort`, which contains a pointer to the port interpreted by the Window Manager as the Window Manager port.

GetCWMgrPort

Gets a pointer to the Window Manager port on a system that supports Color QuickDraw.

```
pascal void GetCWMgrPort (CGrafPtr *wMgrCPort);
```

`wMgrCPort` A pointer to an address at which `GetCWMgrPort` places a pointer to the Window Manager port.

DESCRIPTION

The `GetCWMgrPort` function places a pointer to the color Window Manager port in the value referenced through the parameter `wMgrCPort`. The `GetCWMgrPort` function is available only on computers with Color QuickDraw.

The Window Manager port is a graphics port that occupies all of the main screen. Ordinarily, your application doesn't need to access the Window Manager port.

Note

Do not change any regions of the Window Manager port. If you do, the Window Manager might not handle overlapping windows properly. ♦

GetWMgrPort

Gets a pointer to the Window Manager port on a system with only the original monochrome QuickDraw.

```
pascal void GetWMgrPort (GrafPtr *wPort);
```

`wPort`

A pointer to an address at which `GetWMgrPort` places a pointer to the Window Manager port.

DESCRIPTION

The `GetWMgrPort` function places a pointer to the Window Manager port in the value referenced through the parameter `wPort`.

The Window Manager port is a graphics port that occupies all of the main screen. Ordinarily, your application doesn't need to access the Window Manager port.

Note

Do not change any regions of the Window Manager port. If you do, the Window Manager might not handle overlapping windows properly. ♦

Manipulating Window Color Information

This section describes the functions you use for setting and retrieving window color information. Your application does not normally change window color information.

SetWinColor

Sets a window's window color table.

```
pascal void SetWinColor (WindowPtr theWindow,
                        WCTabHandle newColorTable);
```

`theWindow` A pointer to the window's window record.

`newColorTable` A handle to a window color table record, which defines the colors for the window's new color table.

DESCRIPTION

The `SetWinColor` function sets a window's color table. If the window has no auxiliary window record, it creates a new one with the specified window color table and adds it to the auxiliary window list. If the window already has an auxiliary record, its window color table is replaced. The Window Manager then redraws the window frame and highlighted text in the new colors and sets the window's background color to the new content color.

If the new color table has the same entries as the default color table, `SetWinColor` changes the auxiliary window record so that it points to the default color table.

Window color table resources (resources of type 'wctb') should not be purgeable.

If you specify a value of `nil` for the parameter `theWindow`, `SetWinColor` changes the default color table in memory. Your application shouldn't, however, change the default color table.

SEE ALSO

For a description of a window color table, see “The Window Color Table Record” (page 8-490). For a description of the auxiliary window record, see “The Auxiliary Window Record” (page 8-492). For a description of the 'wctb' resource, see “The Window Color Table Resource” (page 8-571).

GetAuxWin

Gets a handle to a window's auxiliary window record.

```
pascal Boolean GetAuxWin (WindowPtr theWindow,
                          AuxWinHandle *awHndl);
```

`theWindow` A pointer to the window's window record.

`awHndl` A pointer to an address at which `GetAuxWin` places a handle to the window's auxiliary window record.

DESCRIPTION

The `GetAuxWin` function returns a Boolean value that reports whether or not the window has an auxiliary window record, and it sets the value referenced through the parameter `awHnd1` to contain a handle to the window's auxiliary window record.

If the window has no auxiliary window record, `GetAuxWin` places the default window color table in the value referenced through `awHnd1` and returns a value of `false`.

SEE ALSO

For a description of the auxiliary window record, see “The Auxiliary Window Record” (page 8-492).

Low-Level Functions

This section describes the low-level functions that are called by higher-level Window Manager functions. Ordinarily, you won't need to use these functions.

CheckUpdate

Scans the window list for windows that need updating.

```
pascal Boolean CheckUpdate (EventRecord *theEvent);
```

`theEvent` A pointer to an event record to be filled in if a window needs updating.

DESCRIPTION

The `CheckUpdate` function scans the window list from front to back, checking for a visible window that needs updating (that is, a visible window whose update region is not empty). If it finds one whose window record contains a picture handle, it redraws the window itself and continues through the list. If it finds a window record whose update region is not empty and whose window record does not contain a picture handle, it stores an update event in the event record

referenced through the parameter `theEvent` and returns `true`. If it finds no such window, it returns `false`.

The Event Manager is the only software that ordinarily calls `CheckUpdate`.

ClipAbove

Determines the clip region of the Window Manager port.

```
pascal void ClipAbove (WindowPtr window);
```

`window` A pointer to the window's complete window record.

DESCRIPTION

The `ClipAbove` function sets the clip region of the Window Manager port to be the area of the desktop that intersects the current clip region, minus the structure regions of all the windows in front of the specified window.

The `ClipAbove` function retrieves the desktop region from the global variable `GrayRgn`.

The Window Manager is the only software that ordinarily calls `ClipAbove`.

SaveOld

Saves a window's current structure and content regions.

```
pascal void SaveOld (WindowPtr window);
```

`window` A pointer to the window's complete window record.

DESCRIPTION

The `SaveOld` function saves the specified window's current structure region and content region for the `DrawNew` function (page 8-548). Each call to `SaveOld` must be balanced by a subsequent call to `DrawNew`.

The Window Manager calls `DrawNew` before updating a window.

DrawNew

Erases and updates changed window regions.

```
pascal void DrawNew (WindowPtr window,  
                    Boolean update);
```

`window` A pointer to the window's complete window record.

`update` A Boolean value that determines whether the regions are updated.

DESCRIPTION

The `DrawNew` function erases the parts of a window's structure and content regions that are part of the window's former state and part of its new state but not both.

If the `update` parameter is set to `true`, `DrawNew` also updates the erased regions.

Each call to `SaveOld` must be balanced by a subsequent call to `DrawNew`. In C, `SaveOld` and `DrawNew` can't be nested. That is, you must call `DrawNew` before the next call to `SaveOld`.

The Window Manager is the only software that ordinarily calls `DrawNew`.

ASSEMBLY-LANGUAGE INFORMATION

In assembly language, you can nest `SaveOld` and `DrawNew` if you save and restore the values of the global variables `OldStructure` and `OldContent`.

PaintOne

Redraws the invalid, exposed portions of one window on the desktop.

```
pascal void PaintOne (WindowPtr window, RgnHandle clobberedRgn);
```

Window Manager

`window` A pointer to the window's complete window record.

`clobberedRgn` A handle to the region that has become invalid.

DESCRIPTION

The `PaintOne` function “paints” the invalid portion of the specified window and all windows above it. It draws as much of the window frame as is in `clobberedRgn` and, if some content region is exposed, erases the exposed area (paints it with the background pattern) and adds it to the window's update region. If the value of the `window` parameter is `nil`, the window is the desktop, and `PaintOne` paints it with the desktop pattern.

The Window Manager is the only software that ordinarily calls `PaintOne`.

ASSEMBLY-LANGUAGE INFORMATION

The global variables `SaveUpdate` and `PaintWhite` are flags used by `PaintOne`. Normally both flags are set. Clearing `SaveUpdate` prevents `clobberedRgn` from being added to the window's update region. Clearing `PaintWhite` prevents `clobberedRgn` from being erased before being added to the update region (this is useful, for example, if the background pattern of the window isn't the background pattern of the desktop). The Window Manager sets both flags periodically, so you should clear the appropriate flags each time you need them to be clear.

PaintBehind

Redraws a series of windows in the window list.

```
pascal void PaintBehind (WindowPtr startWindow,
                        RgnHandle clobberedRgn);
```

`startWindow` A pointer to the window's complete window record.

`clobberedRgn` A handle to the region that has become invalid.

DESCRIPTION

The `PaintBehind` function calls `PaintOne` (page 8-548) for `startWindow` and all the windows behind `startWindow`, clipped to `clobberedRgn`.

The Window Manager is the only software that ordinarily calls `PaintBehind`.

ASSEMBLY-LANGUAGE INFORMATION

Because `PaintBehind` clears the global variable `PaintWhite` before calling `PaintOne`, `clobberedRgn` isn't erased. The `PaintWhite` global variable is reset after the call to `PaintOne`.

CalcVis

Calculates the visible region of a window.

```
pascal void CalcVis (WindowPtr window);
```

`window` A pointer to the window's complete window record.

DESCRIPTION

The `CalcVis` function calculates the visible region of the specified window by starting with its content region and subtracting the structure region of each window in front of it.

The Window Manager is the only software that ordinarily calls `CalcVis`.

CalcVisBehind

Calculates the visible regions of a series of windows.

```
pascal void CalcVisBehind (WindowPtr startWindow,  
                           RgnHandle clobberedRgn);
```

`startWindow` A pointer to a window's window record.

`clobberedRgn` A handle to the desktop region that has become invalid.

DESCRIPTION

The `CalcVisBehind` function calculates the visible regions of the window specified by the `startWindow` parameter and all windows behind `startWindow` that intersect `clobberedRgn`. It is called after `PaintBehind`.

The Window Manager is the only software that ordinarily calls `CalcVisBehind`.

Accessors for Low-Memory Globals

You should never access low-memory globals directly. Instead, use a high-level Window Manager function, if one is available. If a high-level function is not available, you can use one of the accessors described here.

LMSetAuxWinHead

Sets the handle used by the Window Manager to identify the first auxiliary window record in the auxiliary window list.

```
pascal void LMSetAuxWinHead (AuxWinHandle value);
```

`value` A handle to the address you want to set for the first auxiliary window record in the auxiliary window list.

DESCRIPTION

The Window Manager creates and maintains the information in auxiliary window records. Your application seldom, if ever needs to access the auxiliary window list.

The `LMSetAuxWinHead` function sets the value of the low-memory global `AuxWinHead`, which contains a handle to the address interpreted by the Window Manager as the first auxiliary window record in the auxiliary window list.

SEE ALSO

For a description of the auxiliary window record, see “The Auxiliary Window Record” (page 8-492). To get a handle to an window’s auxiliary window record, use the `GetAuxWin` function (page 8-545). To get a handle to the first auxiliary window record in the auxiliary window list, use `LMGetAuxWinHead` (page 8-552).

LMGetAuxWinHead

Gets the handle used by the Window Manager to identify the first auxiliary window record in the auxiliary window list.

```
pascal AuxWinHandle LMGetAuxWinHead (void);
```

DESCRIPTION

The Window Manager creates and maintains the information in auxiliary window records. Your application seldom, if ever needs to access the auxiliary window list.

The `LMSetAuxWinHead` function returns the value of the low-memory global `AuxWinHead`, which contains a handle to the address interpreted by the Window Manager as the first auxiliary window record in the auxiliary window list.

SEE ALSO

For a description of the auxiliary window record, see “The Auxiliary Window Record” (page 8-492). To get a handle to an window’s auxiliary window record, use the `GetAuxWin` function (page 8-545). To set a handle to the first auxiliary window record in the auxiliary window list, use `LMSetAuxWinHead` (page 8-551).

LMSetDeskHook

Sets the procedure called by the Window Manager to paint the desktop.

```
pascal void LMSetDeskHook (UniversalProcPtr value);
```


`value` A universal procedure pointer to the procedure you wish to set as the desk hook function.

DESCRIPTION

The `LMSetDeskHook` function sets the value of the low-memory global `DeskHook`, which contains a pointer to the procedure used by the Window Manager as the desk hook function.

SEE ALSO

To get the optional procedure that the Window Manager calls to paint the desktop, use the `LMGetDeskHook` function (page 8-553).

LMGetDeskHook

Gets the optional procedure called by the Window Manager to paint the desktop.

```
pascal UniversalProcPtr LMGetDeskHook (void);
```

DESCRIPTION

The `LMGetDeskHook` function returns the value of the low-memory global `DeskHook`, which contains a pointer to the procedure used by the Window Manager as the desk hook function.

SEE ALSO

To set the optional procedure that the Window Manager calls to paint the desktop, use the `LMSetDeskHook` function (page 8-552).

LMSetDragHook

Sets the optional procedure called by the Window Manager during `TrackGoAway` (page 8-531), `TrackBox` (page 8-529), `DragWindow` (page 8-519), `GrowWindow` (page 8-526), and `DragGrayRgn` (page 8-521) functions.

```
pascal void LMSetDragHook (UniversalProcPtr value);
```

value A universal procedure pointer to the procedure you wish to set as the drag hook function.

DESCRIPTION

The `LMSetDragHook` function sets the value of the low-memory global `DragHook`, which contains a pointer to the procedure interpreted by the Window Manager as the drag hook function.

SEE ALSO

To get the optional drag hook procedure that the Window Manager calls, use the `LMGetDragHook` function (page 8-552).

LMGetDragHook

Gets the optional procedure called by the Window Manager during `TrackGoAway` (page 8-531), `TrackBox` (page 8-529), `DragWindow` (page 8-519), `GrowWindow` (page 8-526), and `DragGrayRgn` (page 8-521) functions.

```
pascal UniversalProcPtr LMGetDragHook (void);
```

DESCRIPTION

The `LMGetDragHook` function returns the value of the low-memory global `DragHook`, which contains a pointer to the procedure used by the Window Manager as the drag hook function.

SEE ALSO

To set the optional drag hook procedure that the Window Manager calls, use the `LMSetDragHook` function (page 8-554).

LMSetDragPattern

Sets the pattern of the dragged region's outline.

```
pascal void LMSetDragPattern (const Pattern *dragPatternValue);
```

`dragPatternValue`

A pointer to the pattern you wish to set as the dragged pattern's outline.

DESCRIPTION

The `LMSetDragPattern` function sets the value of the low-memory global `DragPattern`.

SEE ALSO

To get the pattern of the dragged region's outline, use `LMGetDragPattern` (page 8-555).

LMGetDragPattern

Gets the pattern of the dragged region's outline.

```
pascal void LMGetDragPattern (Pattern *dragPatternValue);
```

`dragPatternValue`

A pointer to the address at which the `LMGetDragPattern` function returns the pattern used by the Window Manager as the dragged pattern's outline.

DESCRIPTION

The `LMSetDragPattern` function gets the value of the low-memory global `DragPattern`.

SEE ALSO

To set the pattern of the dragged region's outline, use `LMSetDragPattern` (page 8-555).

LMSetOldContent

Sets the handle used by the Window Manager to identify the saved content region.

```
pascal void LMSetOldContent (RgnHandle value);
```

`value` A handle to the region you wish to set as the saved content region.

DESCRIPTION

The `LMSetOldContent` function sets the value of the low-memory global `OldContent`, which the Window Manager interprets as a handle to the saved content region.

SEE ALSO

To get the handle used by the Window Manager to identify the saved content region, use the `LMGetOldContent` function (page 8-557).

LMGetOldContent

Returns the handle used by the Window Manager to identify the saved content region.

```
pascal RgnHandle LMGetOldContent (void);
```

DESCRIPTION

The `LMGetOldContent` function returns the value of the low-memory global `OldContent`, which the Window Manager interprets as a handle to the saved content region.

SEE ALSO

To set handle used by the Window Manager to identify the saved content region, use `LMSetOldContent` (page 8-556).

LMSetOldStructure

Sets the handle used by the Window Manager to identify the saved structure region.

```
pascal void LMSetOldStructure (RgnHandle value);
```

<code>value</code>	A handle to the region you wish to set as the saved structure region.
--------------------	---

DESCRIPTION

The `LMSetOldStructure` function sets the value of the low-memory global `OldStructure`, which the Window Manager interprets as a handle to the saved structure region.

SEE ALSO

To get handle used by the Window Manager to identify the saved structure region, use `LMGetOldStructure` (page 8-558).

LMGetOldStructure

Sets the handle used by the Window Manager to identify the saved structure region.

```
pascal RgnHandle LMGetOldStructure (void);
```

DESCRIPTION

The `LMGetOldStructure` function returns the value of the low-memory global `OldStructure`, which the Window Manager interprets as a handle to the saved structure region.

SEE ALSO

To set handle used by the Window Manager to identify the saved content region, use `LMSetOldStructure` (page 8-557).

LMSetPaintWhite

Sets the flag used by the Window Manager to determine whether to paint the window white before an update event.

```
pascal void LMSetPaintWhite (SInt16 value);
```

value

SEE ALSO

To get the flag used by the Window Manager to determine whether to paint the window white before an update event, use the `LMGetPaintWhite` function (page 8-559).

LMGetPaintWhite

Returns the flag used by the Window Manager to determine whether to paint the window white before an update event.

```
pascal SInt16 LMGetPaintWhite ( void );
```

SEE ALSO

To set the flag used by the Window Manager to determine whether to paint the window white before an update event, use the `LMSetPaintWhite` function (page 8-558).

LMSetSaveUpdate

Sets the flag used by the Window Manager to determine whether to generate update events.

```
pascal void LMSetSaveUpdate (SInt16 value);
```

```
value
```

SEE ALSO

To get the flag used by the Window Manager to determine whether to generate update events, use the `LMGetSaveUpdate` function (page 8-560).

LMGetSaveUpdate

Returns the flag used by the Window Manager to determine whether to generate update events.

```
pascal SInt16 LMGetSaveUpdate (void);
```

DESCRIPTION

The `LMGetSaveUpdate` function sets the value of the low-memory global `SaveUpdate`, which the Window Manager uses to determine whether to generate update events.

SEE ALSO

To set the flag used by the Window Manager to determine whether to generate update event, use the `LMSetSaveUpdate` function (page 8-559).

LMSetSaveVisRgn

Sets the handle used by the Window Manager to identify the saved visible region.

```
pascal void LMSetSaveVisRgn (RgnHandle value);
```

<code>value</code>	The region you want the Window Manager to interpret as the saved visible region.
--------------------	--

DESCRIPTION

The `LMSetSaveVisRgn` function sets the value of the low-memory global `SaveVisRgn`, which the Window Manager interprets as a handle to the saved visible region.

SEE ALSO

To get the handle used by the Window Manager to identify the saved visible region, use the `LMGetSaveVisRgn` function (page 8-561).

LMGetSaveVisRgn

Returns the handle used by the Window Manager to identify the saved visible region.

```
pascal RgnHandle LMGetSaveVisRgn (void);
```

DESCRIPTION

The `LMGetSaveVisRgn` function returns the value of the low-memory global `SaveVisRgn`, which the Window Manager interprets as a handle to the saved visible region.

SEE ALSO

To set the handle used by the Window Manager to identify the saved visible region, use the `LMSetSaveVisRgn` function (page 8-560).

LMSetWindowList

Sets the pointer used by the Window Manager to identify the first window in the window list.

```
pascal void LMSetWindowList (WindowPtr value);
```

value	A pointer to the window you want to set as the first window in the window list.
-------	---

DESCRIPTION

The `LMSetWindowList` function sets the value of the low-memory global `WindowList`, which the Window Manager interprets as a pointer to the first window in the window list.

SEE ALSO

To get the pointer used by the Window Manager to identify the first window in the window list, use the `LMGetWindowList` function (page 8-562).

LMGetWindowList

Returns the pointer used by the Window Manager to identify the first window in the window list.

```
pascal WindowPtr LMGetWindowList (void);
```

DESCRIPTION

The `LMGetWindowList` function returns the value of the low-memory global `WindowList`, which the Window Manager interprets as a pointer to the first window in the window list.

SEE ALSO

To set the pointer used by the Window Manager to identify the first window in the window list, use the `LMSetWindowList` function (page 8-561).

Application-Defined Function

This section describes the window definition function. The Window Manager supplies window definition functions that handle the standard window types. If your application defines its own window types, you must supply your own window definition function to handle them.

Window Definition Function

Store your definition function as a resource of type 'WDEF' with an ID from 128 through 4096. (Window definition function resource IDs 0 and 1 are the standard window definition functions; resource ID 124 is the standard floating window definition function; resource IDs 2 through 123 and 125 through 127 are reserved by Apple Computer, Inc.)

Your window definition function can support up to 16 variation codes, which are identified by integers 0 through 15. To invoke your own window type, you specify the window's definition ID, which contains the resource ID of the window's definition function in the upper 12 bits and the variation code in the lower 4 bits. Thus, for a given resource ID and variation code, the window definition ID is

$(16 * \text{resource ID}) + (\text{variation code})$

When you create a window, the Window Manager calls the Resource Manager to access the window definition function. The Resource Manager reads the window definition function into memory and returns a handle to it. The Window Manager stores this handle in the `windowDefProc` field of the window record. (If 24-bit addressing is in effect, the Window Manager stores the variation code in the lower 4 bits of the `windowDefProc` field; if 32-bit addressing is in effect, the Window Manager stores the variation code elsewhere.) Later, when it needs to perform a type-dependent action on the window, the Window Manager calls the window definition function and passes it the variation code as a parameter.

The Window Manager defines the window definition function and related constants as described in “Window Definition Function Type and Macros” (page 8-494) and “Window Definition Function Enumerators” (page 8-495). For information about creating your own window definition function, see the description of the `MyWindow` function (page 8-563).

MyWindow

The window definition function is responsible for drawing the window frame, reporting the region where mouse-down events occur, calculating the window's structure region and content region, drawing the size box, resizing the window frame when the user drags the size box, and performing any customized initialization or disposal tasks.

You can give your window definition function any name you wish. As described in “Window Definition Function Type and Macros” (page 8-494), it takes four parameters and returns a result code:

```
pascal long MyWindow (short varCode,
                      WindowPtr theWindow,
                      short message,
                      long param);
```

<code>varCode</code>	The window's variation code.
<code>theWindow</code>	A pointer to the window's window record.
<code>message</code>	A code for the task to be performed. The <code>message</code> parameter contains one of the task codes defined in “Window Definition Function Enumerators” (page 8-495). The subsections that follow explain each of these tasks in detail.
<code>param</code>	Data associated with the task specified by the <code>message</code> parameter. If the task requires no data, this parameter is ignored.

Your window definition function should perform whatever task is specified by the `message` parameter and return a function result if appropriate. If the task performed requires no result code, return 0.

The function's entry point must be at the beginning of the function.

You can set up the various tasks as subroutines inside the window definition function, but you're not required to do so.

Drawing the Window Frame

When you receive a `wDraw` message, draw the window frame in the current graphics port, which is the Window Manager port.

You must make certain checks to determine exactly how to draw the frame. If the value of the `visible` field in the window record is `false`, you should do nothing; otherwise, you should examine the `param` parameter and the status flags in the window record:

- If the value of `param` is 0, draw the entire window frame.
- If the value of `param` is 0 and the `hilited` field in the window record is `true`, highlight the frame to show that the window is active.

- If the value of the `goAwayFlag` field in the window record is also `true`, draw a close box in the window frame.
- If the value of the `spareFlag` field in the window record is also `true`, draw a zoom box in the window frame.
- If the value of the `param` parameter is `wInGoAway`, add highlighting to, or remove it from, the window's close box.
- If the value of the `param` parameter is `wInZoom`, add highlighting to, or remove it from, the window's zoom box.

Note

When the Window Manager calls a window definition function with a message of `wDraw`, it stores a value of type `short` in the `param` parameter without clearing the high-order word. When processing the `wDraw` message, use only the low-order word of the `param` parameter. ♦

The window frame typically but not necessarily includes the window's title, which should be displayed in the system font and system font size. The Window Manager port is already set to use the system font and system font size.

When designing a title bar that includes the window title, allow at least 16 pixels vertically to support localization for script systems in which the system font can be no smaller than 12 points.

Note

Nothing drawn outside the window's structure region is visible. ♦

Returning the Region of a Mouse-Down Event

When you receive a `wHit` message, you must determine where the cursor was when the mouse button was pressed. The `wHit` message is accompanied by the mouse location, in global coordinates, in the `param` parameter. The vertical coordinate is in the high-order word of the parameter, and the horizontal coordinate is in the low-order word. You return one of the hit return codes defined in "Window Definition Function Enumerators" (page 8-495).

The return value `wNoHit` might mean (but not necessarily) that the point isn't in the window. The standard window definition functions, for example, return `wNoHit` if the point is in the window frame but not in the title bar.

Return the constants `wInGrow`, `wInGoAway`, `wInZoomIn`, and `wInZoomOut` only if the window is active—by convention, the size box, close box, and zoom box aren't drawn if the window is inactive. In an inactive document window, for example, a mouse-down event in the part of the title bar that would contain the close box if the window were active is reported as `wInDrag`.

Calculating Regions

When you receive the `wCalcRgn` message, you calculate the window's structure and content regions based on the current graphics port's port rectangle. These regions, whose handles are in the `strucRgn` and `contRgn` fields of the window record, are in global coordinates. The Window Manager requests this operation only if the window is visible.

▲ WARNING

When you calculate regions for your own type of window, do not alter the clip region or the visible region of the window's graphics port. The Window Manager and QuickDraw take care of this for you. Altering the clip region or visible region may damage other windows. ▲

Initializing a New Window

When you receive the `wNew` message, you can perform any type-specific initialization that may be required. If the content region has an unusual shape, for example, you might allocate memory for the region and store the region handle in the `dataHandle` field of the window record. The initialization function for a standard document window creates the `wStateData` record for storing zooming data.

Disposing of a Window

When you receive the `wDispose` message, you can perform any additional tasks necessary for disposing of a window. You might, for example, release memory that was allocated by the initialization function. The dispose function for a standard document window disposes of the `wStateData` record.

Resizing a Window

When you receive the `wGrow` message, draw a grow image of the window. With the `wGrow` message you receive a pointer to a rectangle, in global coordinates,

whose upper-left corner is aligned with the port rectangle of the window's graphics port. Your grow image should fit inside the rectangle. As the user drags the mouse, the Window Manager sends repeated `wGrow` messages, so that you can change your grow image to match the changing mouse location.

Draw the grow image in the current graphics port, which is the Window Manager port, in the current pen pattern and pen mode. These are set up (as `gray` and `notPatXor`) to conform to the Macintosh user interface guidelines.

The grow function for a standard document window draws a dotted (gray) outline of the window and also the lines delimiting the title bar, size box, and scroll bar areas.

Drawing the Size Box

When you receive the `wDrawGIcon` message, you draw the size box in the content region if the window is active—if the window is inactive, draw whatever is appropriate to show that the window cannot currently be sized.

Note

If the size box is located in the window frame instead of the content region, do nothing in response to the `wDrawGIcon` message, instead drawing the size box in response to the `wDraw` message. ♦

The function that draws a size box for an active document window draws the size box in the lower-right corner of the port rectangle of the window's graphics port. It also draws lines delimiting the size box and scroll bar areas. For an inactive document window, it erases the size box and draws the delimiting lines.

Resources

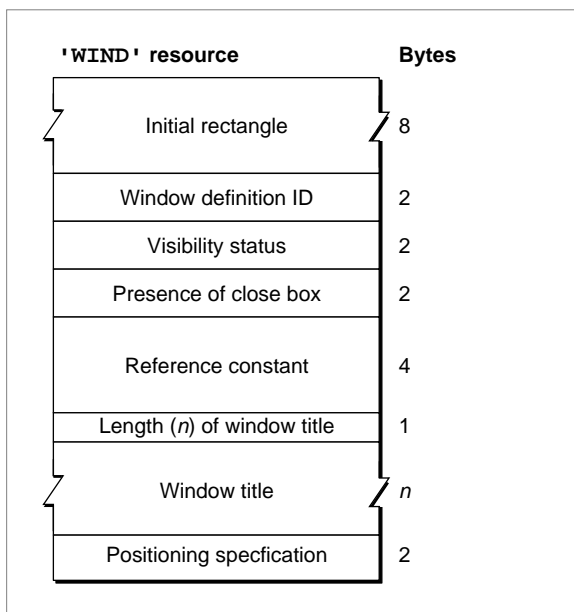
This section describes the resources used by the Window Manager:

- the 'WIND' resource, used for describing the characteristics of windows
- the 'WDEF' resource, which holds a window definition function
- the 'wctb' resource, which defines the colors to be used for a window's frame and highlighting

The Window Resource

You typically define a window resource for each type of window that your application creates. Figure 7-3 illustrates a compiled 'WIND' resource.

Figure 7-3 Structure of a compiled window ('WIND') resource



A compiled version of a window resource contains the following elements:

- The upper-left and lower-right corners, in global coordinates, of a rectangle that defines the initial size and placement of the window's content region. Your application can change this rectangle before displaying the window, either programmatically or through an optional positioning code described later in this section.
- The window's definition ID, which incorporates both the resource ID of the window definition function that will handle the window and an optional variation code. Together, the window definition function resource ID and the variation code define a window type. Place the resource ID of the window

definition function in the upper

12 bits of the definition ID. Window definition functions with IDs 0 through 127 are reserved for use by Apple Computer, Inc. Place the optional variation code in the lower 4 bits of the definition ID.

If you're using one of the standard window types, the definition ID is one of the enumerators described in "Window Definition IDs" (page 8-476).

- A specification that determines whether the window is visible or invisible. This characteristic controls only whether the Window Manager displays the window, not necessarily whether the window can be seen on the screen. (A visible window entirely covered by other windows, for example, is "visible" even though the user cannot see it.) You typically create a new window in an invisible state, build the content area of the window, and then display the completed window.
- A specification that determines whether or not the window has a close box. The Window Manager draws the close box when it draws the window frame. The window type specified in the second field determines whether a window can support a close box; this field determines whether the close box is present.
- A reference constant, which your application can use for whatever data it needs to store. When it builds a new window record, the Window Manager stores, in the `refCon` field, whatever value you specify in the fifth element of the window resource. You can also put a place holder here and then set the `refCon` field yourself with the `SetWRefCon` function (page 8-539).
- A string that specifies the window title. The first byte of the string specifies the length of the string (that is, the number of characters in the title plus 1 byte for the length), in bytes.

- An optional positioning specification that overrides the window position established by the rectangle in the first field. The positioning value can be one of the integers defined by the enumerators listed here.:

center	Centered both horizontally and vertically, relative either to a screen or to another window (if a window to be centered relative to another window is wider than the window that preceded it, it is pinned to the left edge; a narrower window is centered)
stagger	Located 10 pixels to the right and 10 pixels below the upper-left corner of the last window (in the case of staggering relative to a screen, the first window is placed just below the menu bar at the left edge of the screen, and subsequent windows are placed on that screen relative to the first window)
alert position	Centered horizontally and placed in the “alert position” vertically, that is, with about one-fifth of the window or screen above the new window and the rest below
parent window	The window in which the user was last working

The seventh element of the resource can contain one of the values specified by these enumerators:

```
enum {
    noAutoCenter          = 0x0000,    /* use initial location */
    centerMainScreen      = 0x280A,    /* center on main screen */
    alertPositionMainScreen = 0x300A,    /* place in alert */
                                /* position on main screen */
    staggerMainScreen     = 0x380A,    /* stagger on main screen */
    centerParentWindow    = 0xA80A,    /* center on parent window */
    alertPositionParentWindow = 0xB00A,    /* place in alert position
                                /* on parent window */
    staggerParentWindow   = 0xB80A,    /* stagger relative
                                /* to parent window */
    centerParentWindowScreen = 0x680A,    /* center on parent
                                /* window screen */
    alertPositionParentWindowScreen = 0x700A,    /* place in alert position
                                /* on parent window screen */
}
```

```

    staggerParentWindowScreen    /* stagger on parent
                                = 0x780A    /* window screen */
};

```

The positioning constants are convenient when the user is creating new documents or when you are handling your own dialog boxes and alert boxes. When you are creating a new window to display a previously saved document, however, you should display the new window in the same rectangle as the previous window (that is, the window the document occupied when it was last saved).

Use the `GetNewCWindow` (page 8-498) or `GetNewWindow` (page 8-501) function to read a 'WIND' resource. Both functions create a new window record and fill it in according to the values specified in a 'WIND' resource.

The Window Definition Function Resource

Window definition functions are stored as resources of type 'WDEF'. The 'WDEF' resource is simply the executable code for the window definition function.

IMPORTANT

All resources of type 'WDEF' should be nonpurgeable. This ensures that the resource is always available, even if your application is not frontmost. ▲

The two standard window definition functions supplied by the Window Manager use resource IDs 0 and 1.

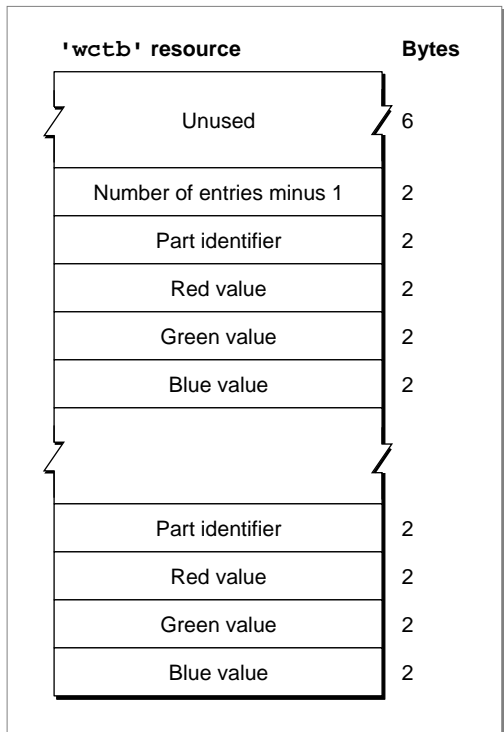
The Window Color Table Resource

You can specify your own window color tables as resources of type 'wctb'.

Ordinarily, you should not define your own window color tables, unless you have some extraordinary need to control the color of a window's frame or text highlighting. To assign a table to a window when you create the window, provide a window color table ('wctb') resource with the same resource ID as the 'WIND' resource from which you create the window.

The window color table resource is an exact image of the window color table data structure. Figure 7-4 illustrates the contents of a compiled 'wctb' resource.

Figure 7-4 Structure of a compiled window color table ('wctb') resource



A compiled version of a window resource contains the following elements:

- An unused field 6 bytes long.
- An integer that specifies the number of entries in the resource (that is, the number of color specification records) minus 1.
- A series of color specification records, each of which consists of a 2-byte part identifier and three 2-byte color values. The part identifier is an integer specified by one of the enumerators listed in “Part Identifiers for ColorSpec Records” (page 8-482).

Color QuickDraw

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

Color QuickDraw

- 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}
```

Color QuickDraw

```

    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.

Color QuickDraw

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 8-1. Some aspects of its contents are discussed after the figure; see page 9-618 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 8-1 The color graphics port

device	Device-specific information
portPixMap	Handle to a pixel map
portVersion	Flags
grafVars	Handle to additional color fields
chExtra	Extra width added to nonspace characters
pnLocHFrac	Fractional horizontal pen position
portRect	Port rectangle
visRgn	Visible region
clipRgn	Clipping region
bkPixPat	Background pattern
rgbFgColor	Requested foreground color
rgbBkColor	Requested background color
pnLoc	Pen location
pnSize	Pen size
pnMode	Pattern mode
pnPixPat	Pen pattern
fillPixPat	Fill pattern
pnVis	Pen visibility
txFont	Font number for text
txFace	Text font style
txMode	Text source mode
txSize	Font size for text
spExtra	Extra width added to space characters
fgColor	Actual foreground color
bkColor	Actual background color
colrBit	Color bit (reserved)
	
grafProcs	Pointer to low-level drawing routines

Color QuickDraw

Table 8-3 on page 9-634 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.

Color QuickDraw

The representation of a color image in memory is a **pixel image**, analogous to the bit image used by basic QuickDraw. A `PixelFormat` record includes a pointer to a pixel image, its dimensions, storage format, depth, resolution, and color usage. The pixel map is diagrammed in Figure 8-2. Some aspects of its contents are discussed after the figure; see page 9-616 for a complete description of its fields.

Figure 8-2 The pixel map

<code>baseAddr</code>	Pointer to the image data
<code>rowBytes</code>	Flags, and bytes in a row
<code>bounds</code>	Boundary rectangle
<code>pmVersion</code>	Pixel map version number
<code>packType</code>	Packing format
<code>packSize</code>	Size of data in packed state
<code>hRes</code>	Horizontal resolution in dots per inch
<code>vRes</code>	Vertical resolution in dots per inch
<code>pixelType</code>	Format of pixel image
<code>pixelSize</code>	Physical bits per pixel
<code>cmpCount</code>	Number of components in each pixel
<code>cmpSize</code>	Number of bits in each component
<code>planeBytes</code>	Offset to next plane
<code>pmTable</code>	Handle to a color table for this image
<code>pmReserved</code>	Reserved

The `baseAddr` field of a `PixelFormat` 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 9-583.

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.)

Color QuickDraw

You can create color tables using either `ColorTable` records (described on page 9-626) or color table ('clut') resources (described on page 9-674). 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 `FillRect`, `FillRoundRect`, `FillArc`, `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.

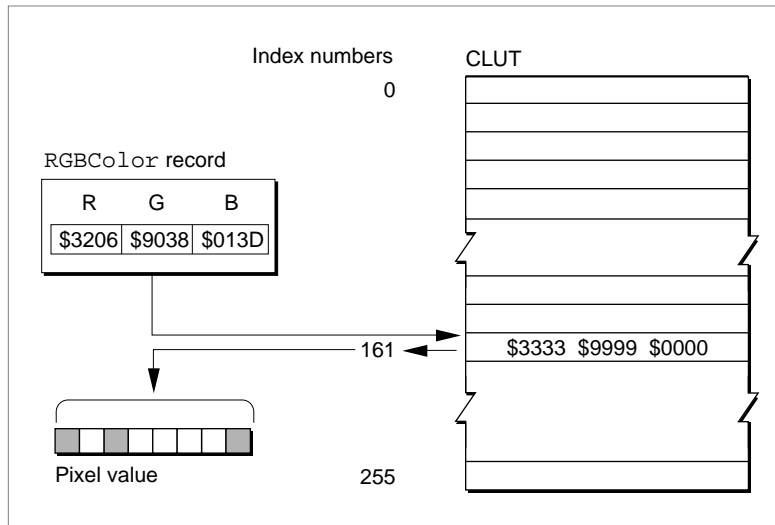
Indexed devices were introduced to support—with minimal memory requirements—the color capabilities of the Macintosh II computer. The pixel value for any color on an indexed device is represented by a single byte. Each byte contains an index number that specifies one of 256 colors available on the device's 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.

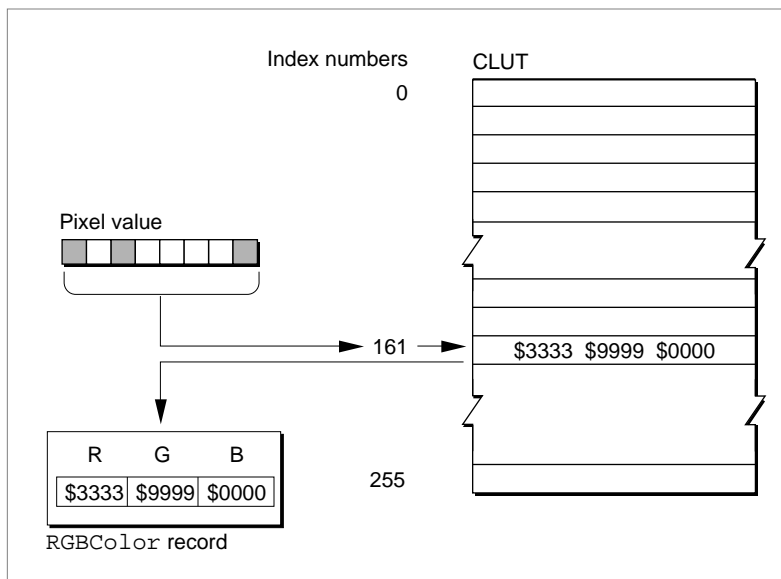
Color QuickDraw

For example, the `RGBColor` record for a medium green pixel is represented on the left side of Figure 8-3. 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 8-3 Translating a 48-bit `RGBColor` record to an 8-bit pixel value on an indexed device

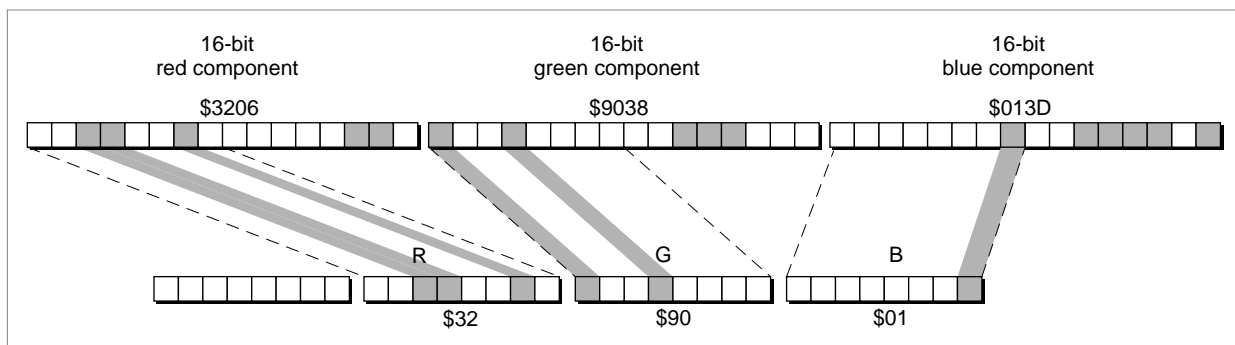


The application might later use the `GetCPixel` procedure to determine the color of a particular pixel. As shown in Figure 8-4, the Color Manager uses the index number stored as the pixel value to find the 48-bit `RGBColor` record stored in the CLUT for that pixel's color—which, as with the medium green in this example, is not necessarily the exact color first specified by the application. The difference, however, is imperceptible.

Figure 8-4 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 8-5, 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 8-5 Translating a 48-bit `RGBColor` record to a 32-bit pixel value on a direct device

Color QuickDraw

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 8-6.

Figure 8-6 Translating a 48-bit `RGBColor` record to a 16-bit pixel value on a direct device

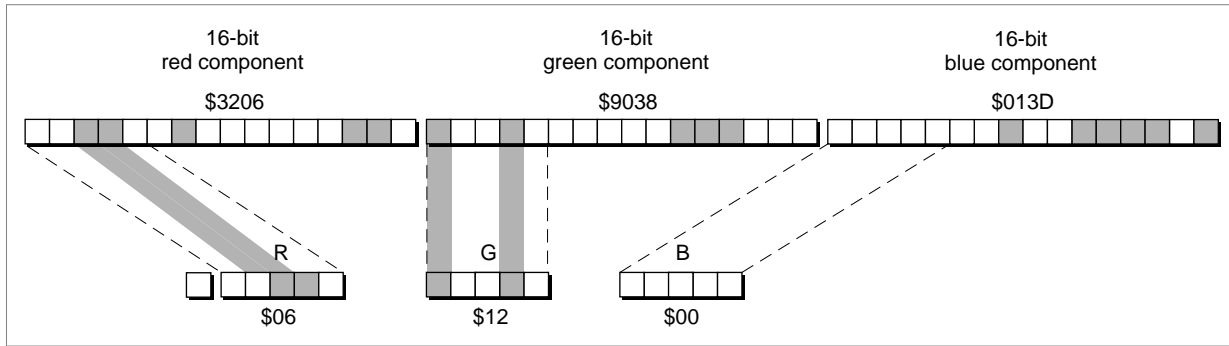


Figure 8-7 shows how Color QuickDraw expands a 32-bit pixel value to a 48-bit `RGBColor` record by dropping the unused high byte of the pixel value and doubling each of its 8-bit components. Note that the resulting 48-bit value differs in the least significant 8 bits of each component from the original `RGBColor` record in Figure 8-5.

Figure 8-7 Translating a 32-bit pixel value to a 48-bit `RGBColor` record

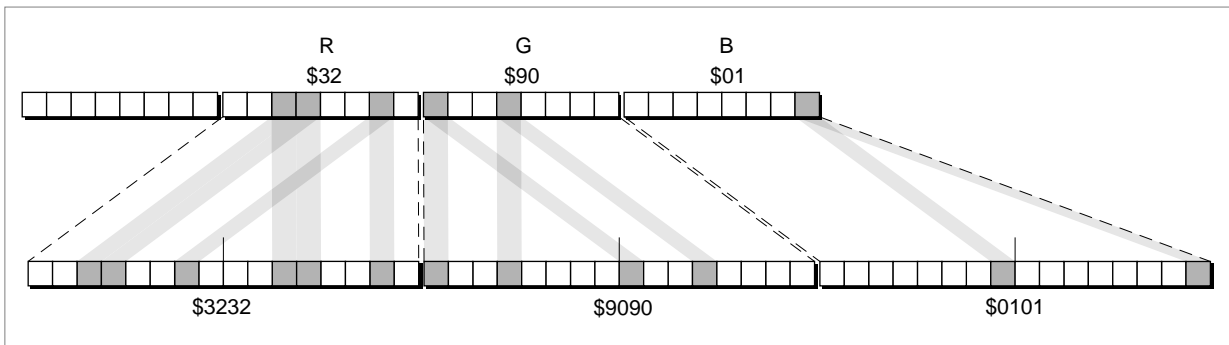
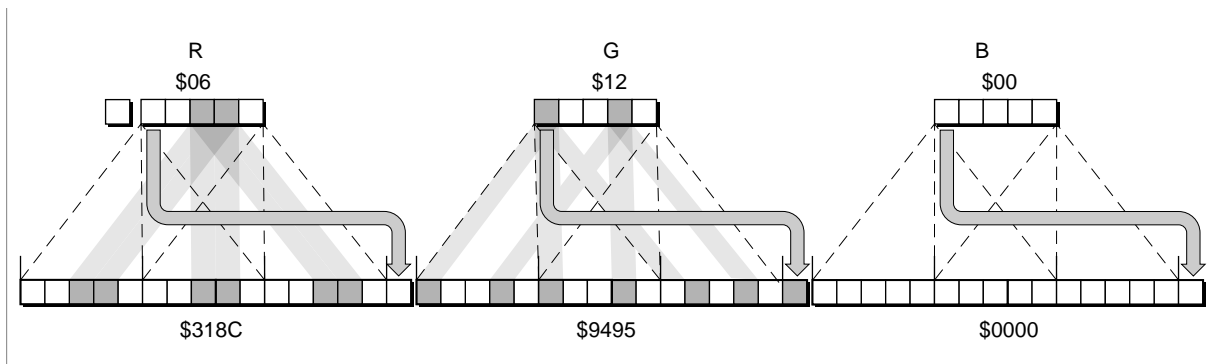


Figure 8-8 shows how Color QuickDraw expands a 16-bit pixel value to a 48-bit `RGBColor` record by dropping the unused high bit of the pixel value and inserting three copies of each 5-bit component and a copy of the most significant bit into each 16-bit field of the `RGBColor` record. Note that the result differs (in the least significant 11 bits of each component) from the original 48-bit value in Figure 8-5. The difference, however, is imperceptible.

Figure 8-8 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 9-662, 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}
```

Color QuickDraw

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 8-1 shows a simplified application-defined procedure called `DoNew` that uses the Window Manager function `GetNewCWindow` to create a color graphics port.

Listing 8-1 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 8-1.

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 9-593, 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. ♦

Color QuickDraw

To specify a foreground color, create an `RGBColor` record. Listing 8-2 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 8-2 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 8-2, 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.

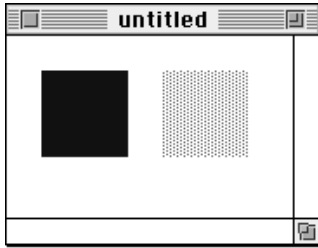
After using `SetRect` to create a rectangle, Listing 8-2 calls `PaintRect` to paint the rectangle. By default, the foreground color is black; by changing the foreground color to dark blue, every pixel that would normally be painted in black is instead painted in dark blue.

Listing 8-2 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 8-2 are illustrated in the grayscale screen capture shown in Figure 8-9.

Figure 8-9 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 `FillRect`, `FillRoundRect`, `FillOval`, `FillArc`, `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 `FillRect`, `FillRoundRect`, `FillOval`, `FillArc`, `FillCPoly`, and `FillCRgn` procedures.

Note

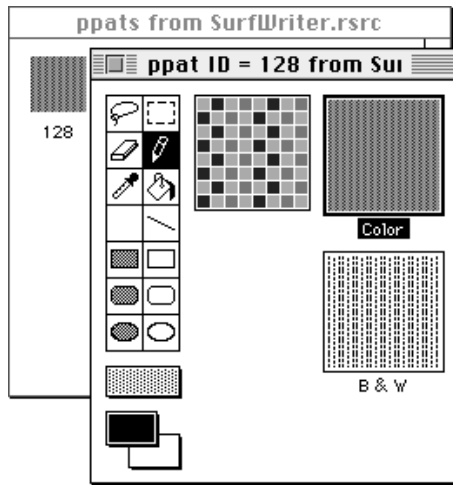
Because a pixel pattern already contains color, Color QuickDraw ignores the foreground and background colors when your application uses these routines to draw with a pixel pattern. Color QuickDraw also ignores the pen mode by drawing the pixel pattern directly onto the pixel image. ♦

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."

Color QuickDraw

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 8-10 illustrates a ResEdit window displaying an application's 'ppat' resource with resource ID 128.

Figure 8-10 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 8-3 shows the Rez input created from the 'ppat' resource created in Figure 8-10.

Listing 8-3 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"
};
```

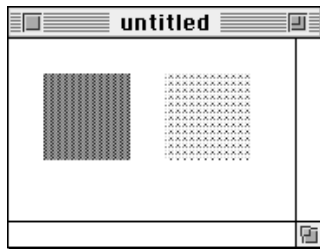
To retrieve the pixel pattern stored in a 'ppat' resource, you can use the `GetPixPat` function. Listing 8-4 uses `GetPixPat` to retrieve the 'ppat' resource created in Listing 8-3. To assign this pixel pattern to the graphics pen, Listing 8-4 uses the `PenPixPat` procedure.

Listing 8-4 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 8-4 uses the `PaintRect` procedure to draw a rectangle. The rectangle on the left side of Figure 8-11 illustrates the effect of painting a rectangle with the previously defined pen pattern.

Figure 8-11 Painting and filling rectangles with pixel patterns



Color QuickDraw

The rectangle on the right side of Figure 8-11 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

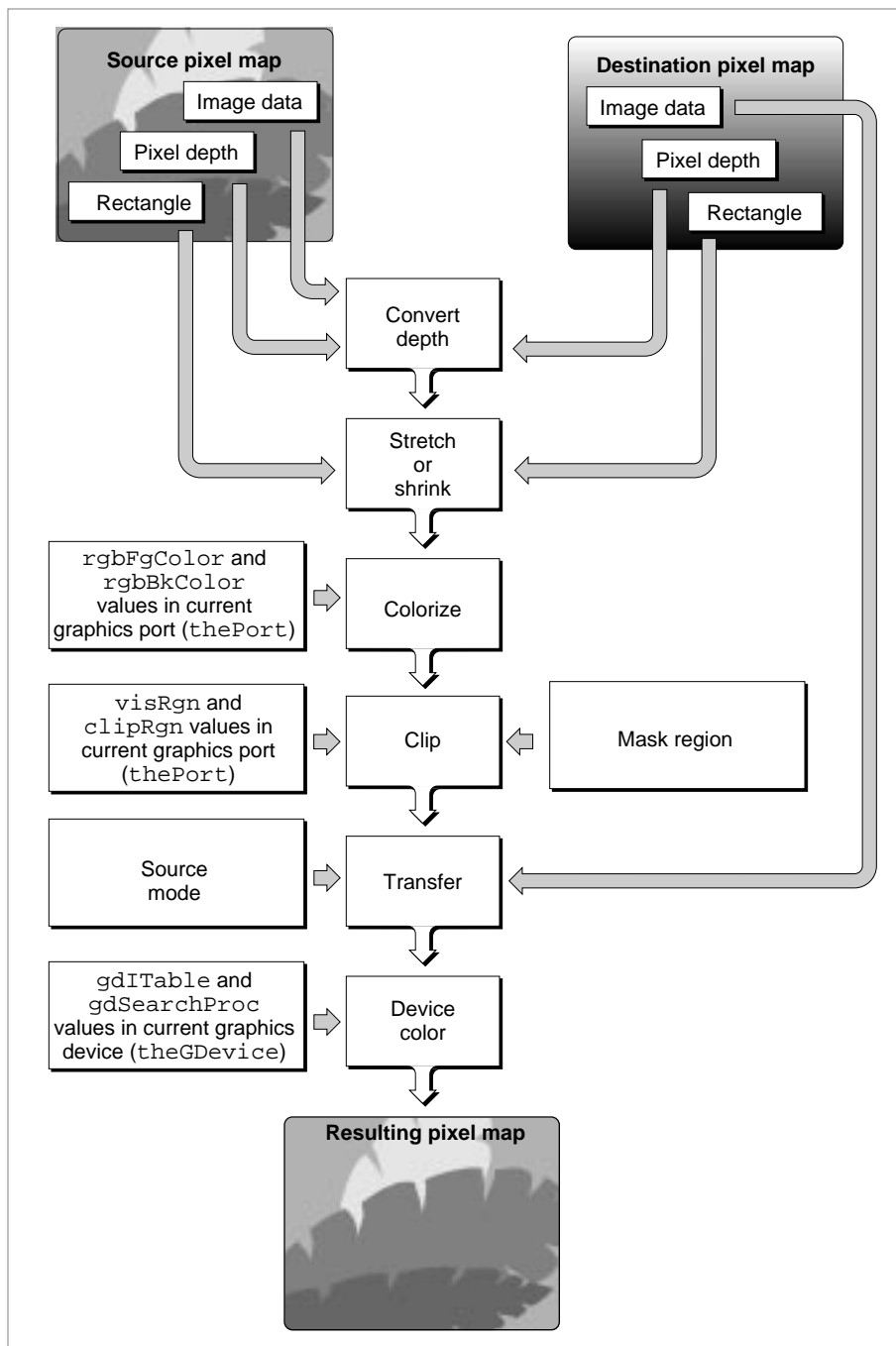
As explained in the chapter “QuickDraw Drawing,” QuickDraw has three primary image-processing routines.

- 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.

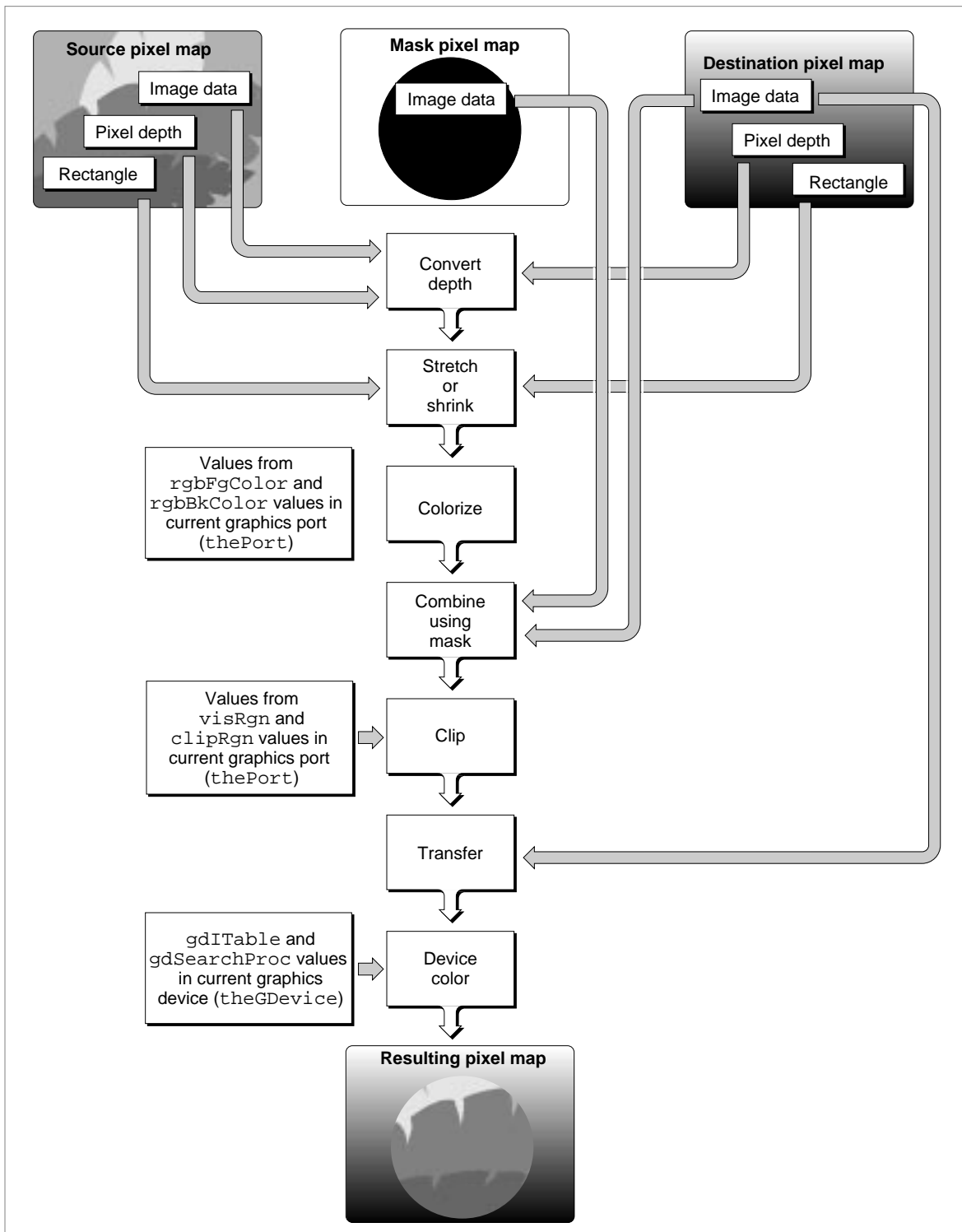
When using `CopyBits`, `CopyMask`, and `CopyDeepMask` to copy images between color graphics ports, you must coerce each port's `CGrafPtr` data type to a `GrafPtr` data type, dereference the `portBits` fields of each, and then pass these “bitmaps” in the `srcBits` and `dstBits` parameters. If your application 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 8-12. Listing 10-1 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 8-12 Copying pixel images with the `CopyBits` procedure

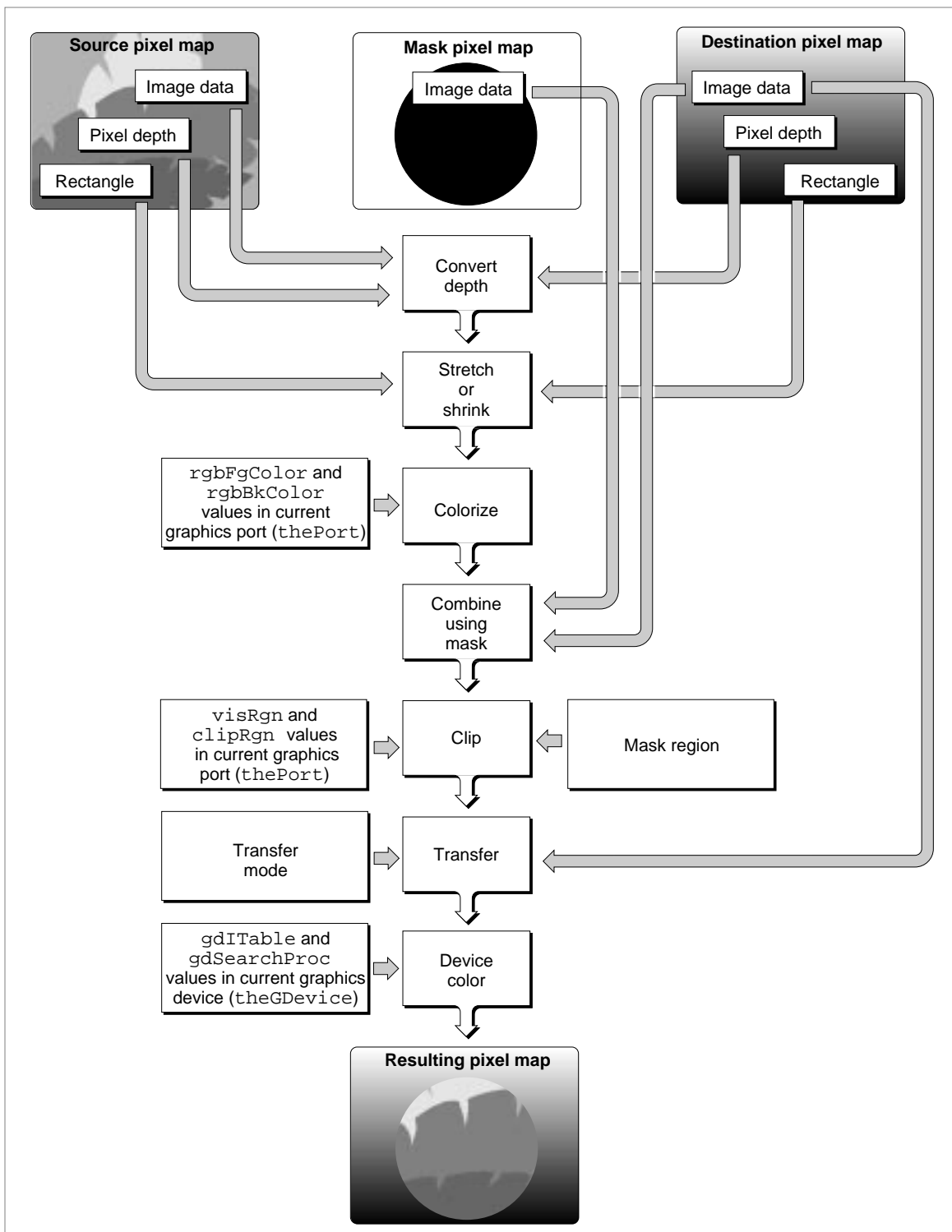
Color QuickDraw

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 8-13, and an example of the effect can be seen in Plate 3 at the front of this book. Listing 10-2 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 8-13 Copying pixel images with the `CopyMask` procedure

Color QuickDraw

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 8-14.

Figure 8-14 Copying pixel images with the `CopyDeepMask` procedure

Color QuickDraw

On indexed devices, pixel images are always copied using the color table of the source `PixelFormat` 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 `PixelFormat` record is ignored. As explained in the chapter “Offscreen Graphics Worlds,” if you need to copy to an offscreen `PixelFormat` 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 `PixelFormat` 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 `PixelFormat` records with greater pixel depths than 1, Color QuickDraw takes a weighted average between the colors of the source and destination `PixelFormat` records. Within each pixel, the calculation is done in RGB color, on a color component basis. A gray `PixelFormat` 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 8-1. 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 9-608.

Table 8-1 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
<code>srcCopy</code>	Apply foreground color	Apply background color	Apply weighted portions of foreground and background colors
<code>notSrcCopy</code>	Apply background color	Apply foreground color	Apply weighted portions of background and foreground colors
<code>srcOr</code>	Apply foreground color	Leave alone	Apply weighted portions of foreground color
<code>notSrcOr</code>	Leave alone	Apply foreground color	Apply weighted portions of foreground color
<code>srcXor</code>	Invert (undefined for colored destination pixel)	Leave alone	Leave alone
<code>notSrcXor</code>	Leave alone	Invert (undefined for colored destination pixel)	Leave alone
<code>srcBic</code>	Apply background color	Leave alone	Apply weighted portions of background color
<code>notSrcBic</code>	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.

Color QuickDraw

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 8-5 shows the code that produced these two pixel maps.

Listing 8-5 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}
```

Color QuickDraw

```

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 8-5 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 8-5 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.

Color QuickDraw

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 `PixelFormat` 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 `PixelFormat` 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.

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. ♦

Color QuickDraw

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 9-648. 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:

$$\text{dest} = \text{source} \times \text{weight}/65,535 + \text{destination} \times (1 - \text{weight}/65,535)$$

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 8-2. (The `hilite` mode is explained in the next section.)

Table 8-2 Arithmetic modes in a 1-bit environment

Initial arithmetic mode	Resulting source mode
<code>blend</code>	<code>srcCopy</code>

Table 8-2 Arithmetic modes in a 1-bit environment

Initial arithmetic mode	Resulting source mode
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 indicate selected text; if the highlight color is yellow, `TextEdit` draws the selected text, then uses `InvertRgn` to produce a yellow background for the text.

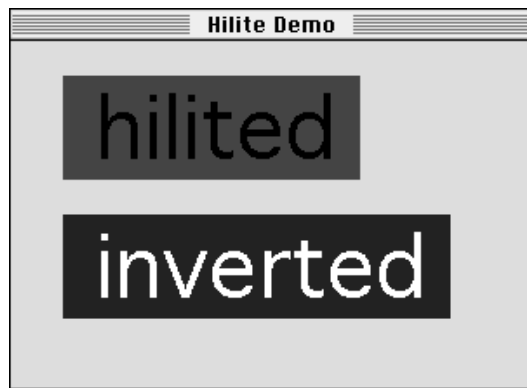
With basic QuickDraw, you can use `InvertRect`, `InvertRgn`, `InvertArc`, `InvertRoundRect`, or `InvertPoly` and any image-copying routine that uses the `srcXor` source mode to invert objects on the screen.

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 8-15 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

Color QuickDraw

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 8-15 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 9-648.

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 8-6 to clear system software's highlight bit (`BitClr` is described in *Inside Macintosh: Operating System Utilities*).

Listing 8-6 Setting the highlight bit

```
PROCEDURE MySetHiliteMode;
BEGIN
    BitClr(Ptr(HiliteMode), pHiliteBit);
END;
```

Listing 8-7 shows the code that produced the effects in Figure 8-15.

Listing 8-7 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

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 `GDevRefCon` 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 9-673.

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.

PixelFormat

A pixel map, which is defined by a data structure of type `PixelFormat`, 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 PixelMap =
RECORD
    baseAddr:      Ptr;          {pixel image}
    rowBytes:      Integer;      {flags, and row width}
    bounds:        Rect;         {boundary rectangle}
    pmVersion:     Integer;      {PixelFormat 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 `PixelFormat` 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 `PixelFormat` record for an offscreen graphics world. Your application should never directly access the **baseAddr** field of the `PixelFormat` record for an offscreen graphics world; instead, your application should always use `GetPixBaseAddr`. ▲

Color QuickDraw

<code>rowBytes</code>	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 <code>rowBytes</code> are used as flags. If bit 15 = 1, the data structure pointed to is a <code>PixMap</code> record; otherwise it is a <code>BitMap</code> record.
<code>bounds</code>	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 <code>gdRect</code> field of the <code>GDevice</code> record for the screen, as described in the chapter "Graphics Devices" in this book.
<code>pmVersion</code>	The version number of Color QuickDraw that created this <code>PixMap</code> record. The value of <code>pmVersion</code> is normally 0. If <code>pmVersion</code> is 4, Color QuickDraw treats the <code>PixMap</code> record's <code>baseAddr</code> field as 32-bit clean. (All other flags are private.) Most applications never need to set this field.
<code>packType</code>	The packing algorithm used to compress image data. Color QuickDraw currently supports a <code>packType</code> of 0, which means no packing, and values of 1 to 4 for packing direct pixels.
<code>packSize</code>	The size of the packed image in bytes. When the <code>packType</code> field contains the value 0, this field is always set to 0.
<code>hRes</code>	The horizontal resolution of the pixel image in pixels per inch. This value is of type <code>Fixed</code> ; by default, the value here is \$00480000 (for 72 pixels per inch).
<code>vRes</code>	The vertical resolution of the pixel image in pixels per inch. This value is of type <code>Fixed</code> ; by default, the value here is \$00480000 (for 72 pixels per inch).
<code>pixelType</code>	The storage format for a pixel image. Indexed pixels are indicated by a value of 0. Direct pixels are specified by a value of <code>RGBDirect</code> , or 16. In the <code>PixMap</code> record of the <code>GDevice</code> record (described in the chapter "Graphics Devices") for a direct device, this field is set to the constant <code>RGBDirect</code> when the screen depth is set.
<code>pixelSize</code>	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.
<code>cmpCount</code>	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.

Color QuickDraw

<code>cmpSize</code>	<p>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 <code>cmpCount</code> field multiplied by the value of the <code>cmpSize</code> field is less than or equal to the value in the <code>pixelSize</code> field.</p> <p>For an indexed pixel value, which has only one component, the value of the <code>cmpSize</code> field is the same as the value of the <code>pixelSize</code> field—that is, 1, 2, 4, or 8.</p> <p>For direct pixels there are two additional possibilities:</p> <p>A 16-bit pixel, which has three components, has a <code>cmpSize</code> value of 5. This leaves an unused high-order bit, which Color QuickDraw sets to 0.</p> <p>A 32-bit pixel, which has three components (red, green, and blue), has a <code>cmpSize</code> value of 8. This leaves an unused high-order byte, which Color QuickDraw sets to 0.</p> <p>If presented with a 32-bit image—for example, in the <code>CopyBits</code> 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 <code>NewHandleClear</code> and <code>NewPtrClear</code>, described in <i>Inside Macintosh: Memory</i>, assist you in allocating prezeroed memory.</p>
<code>planeBytes</code>	The offset in bytes from one drawing plane to the next. This field is set to 0.
<code>pmTable</code>	A handle to a <code>ColorTable</code> record (described on page 9-626) for the colors in this pixel map.
<code>pmReserved</code>	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 9-634. 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`

Color QuickDraw

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}

```

Color QuickDraw

```

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

<code>device</code>	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.
<code>portPixMap</code>	A handle to a <code>PixMap</code> record (described on page 9-616), which describes the pixels in this color graphics port.
<code>portVersion</code>	In the highest 2 bits, flags set to indicate that this is a <code>CGrafPort</code> record and, in the remainder of the field, the version number of Color QuickDraw that created this record.
<code>grafVars</code>	A handle to the <code>GrafVars</code> record (described on page 9-632), which contains additional graphics fields of color information.
<code>chExtra</code>	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 <code>txSize</code> field before it is used. By default, this field contains the value 0.
<code>pnLocHFrac</code>	The fractional horizontal pen position used when drawing text. The value in this field represents the low word of a <code>Fixed</code> number; in decimal, its initial value is 0.5.

<code>portRect</code>	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 <i>content region</i> .) The port rectangle uses the local coordinate system defined by the boundary rectangle in the <code>portPixMap</code> field of the <code>PixMap</code> record. The upper-left corner (which for a window is called the <i>window origin</i>) of the port rectangle usually has a vertical coordinate of 0 and a horizontal coordinate of 0, although you can use the <code>SetOrigin</code> 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.
<code>visRgn</code>	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.
<code>clipRgn</code>	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 <code>ClipRect</code> 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.
<code>bkPixPat</code>	A handle to a <code>PixPat</code> record (described on page 9-628) that describes the background pixel pattern. Procedures such as <code>ScrollRect</code> (described in the chapter "Basic QuickDraw") and <code>EraseRect</code> (described in the chapter "QuickDraw Drawing") use this pattern for filling scrolled or erased areas. Your application can use the <code>BackPixPat</code> procedure (described on page 9-639) to change the background pixel pattern.
<code>rgbFgColor</code>	An <code>RGBColor</code> record (described on page 9-625) that contains the requested foreground color. By default, the foreground color is black, but you can use the <code>RGBForeColor</code> procedure (described on page 9-640) to change the foreground color.
<code>rgbBkColor</code>	An <code>RGBColor</code> record that contains the requested background color. By default, the background color is white, but you can use the <code>RGBBackColor</code> procedure (described on page 9-642) to change the background color.

Color QuickDraw

<code>pnLoc</code>	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."
<code>pnSize</code>	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 <code>PenSize</code> procedure (described in the chapter "QuickDraw Drawing") to change the value in this field.
<code>pnMode</code>	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 9-602.
<code>pnPixPat</code>	A handle to a <code>PixPat</code> record (described on page 9-628) that describes a pixel pattern used like the ink in the graphics pen. Color QuickDraw uses the pixel pattern defined in the <code>PixPat</code> record when you use the <code>Line</code> and <code>LineTo</code> procedures to draw lines with the pen, framing procedures such as <code>FrameRect</code> to draw shape outlines with the pen, and painting procedures such as <code>PaintRect</code> to paint shapes with the pen.
<code>fillPixPat</code>	A handle to a <code>PixPat</code> record (described on page 9-628) that describes the pixel pattern that's used when you call a procedure such as <code>FillRect</code> to fill an area. Notice that this is not in the same location as the <code>fillPat</code> field in a <code>GrafPort</code> record.
<code>pnVis</code>	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."

Color QuickDraw

<code>txFont</code>	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 <i>Inside Macintosh: Text</i> .
<code>txFace</code>	The character style of the text, with values from the set defined by the <code>Style</code> 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 <i>Inside Macintosh: Text</i> .
<code>txMode</code>	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 <code>pnMode</code> 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— <code>srcOr</code> , <code>srcXor</code> , and <code>srcBic</code> —should be used for drawing text. See the chapter “QuickDraw Text” in <i>Inside Macintosh: Text</i> for more information about QuickDraw’s text-handling capabilities.
<code>txSize</code>	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 <i>Inside Macintosh: Text</i> .) The value in this field can be represented by <p style="text-align: center;">$\text{point size} \times \text{device resolution} / 72 \text{ dpi}$</p> <p>where <i>point</i> is a typographical term meaning approximately 1/72 inch.</p>
<code>spExtra</code>	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 <code>spExtra</code> field is useful when a line of characters is to be aligned with both the left and the right margin (sometimes called <i>full justification</i>).
<code>fgColor</code>	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 <code>rgbFgColor</code> field.
<code>bkColor</code>	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 <code>rgbBkColor</code> field.
<code>colrBit</code>	Reserved.
<code>patStretch</code>	A value used during output to a printer to expand patterns if necessary. Your application should not change this value.

Color QuickDraw

<code>picSave</code>	The state of the picture definition. If no picture is open, this field contains <code>NIL</code> ; 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 <code>NIL</code> 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.
<code>rgnSave</code>	The state of the region definition. If no region is open, this field contains <code>NIL</code> ; 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 <code>NIL</code> to disable the region definition, and later restore it to the saved value to resume defining the region.
<code>polySave</code>	The state of the polygon definition. If no polygon is open, this field contains <code>NIL</code> ; 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 <code>NIL</code> to disable the polygon definition, and later restore it to the saved value to resume defining the polygon.
<code>grafProcs</code>	An optional pointer to a <code>CQDProcs</code> record (described on page 9-630) 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 9-669.

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 9-616) 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;
```

Color QuickDraw

Field descriptions

value	The pixel value assigned by Color QuickDraw for the color specified in the <code>rgb</code> 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 <code>RGBColor</code> record (described in the previous section) that fully specifies the color whose approximation Color QuickDraw specifies in the <code>value</code> field.

ColorTable

When creating a `PixMap` record (described on page 9-616) 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 <code>ctSeed</code> 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 <i>Inside Macintosh: Advanced Color Imaging</i> for more information on inverse tables.
--------	---

Color QuickDraw

<code>ctFlags</code>	Flags that distinguish pixel map color tables from color tables in <code>GDevice</code> records (which are described in the chapter “Graphics Devices” in this book).
<code>ctSize</code>	One less than the number of entries in the table.
<code>ctTable</code>	An array of <code>ColorSpec</code> entries, each containing a pixel value and a color specified by an <code>RGBColor</code> 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 9-667.

MatchRec

As described in “Application-Defined Routine” on page 9-671, 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

<code>red</code>	Red value of the seed.
<code>green</code>	Green value of the seed.
<code>blue</code>	Blue value of the seed.
<code>matchData</code>	The value passed in the <code>matchData</code> parameter of the <code>SeedCFill</code> or <code>CalcCMask</code> 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 9-673.

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

<code>patType</code>	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.
<code>patMap</code>	A handle to a <code>PixMap</code> record (described on page 9-616) that describes the pattern's pixel image. The <code>PixMap</code> record can contain indexed or direct pixels.
<code>patData</code>	A handle to the pattern's pixel image.
<code>patXData</code>	A handle to an expanded pixel image used internally by Color QuickDraw.
<code>patXValid</code>	A flag that, when set to -1, invalidates the expanded data.
<code>patXMap</code>	Reserved for use by Color QuickDraw.
<code>pat1Data</code>	A bit pattern (described in the chapter "QuickDraw Drawing") to be used when this pattern is drawn into a <code>GrafPort</code> record (described in the chapter "Basic QuickDraw"). The <code>NewPixPat</code> function (described on page 9-658) 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 9-658), and set it using the `PenPixPat` or `BackPixPat` procedure (described on page 9-637 and page 9-639, 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.

Color QuickDraw

In an RGB pixel pattern, the `patType` field contains the value 2. Using the `MakeRGBPat` procedure (described on page 9-660), 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 9-668.

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 9-666, 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}
```

Color QuickDraw

```

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

<code>rgbOpColor</code>	The color for the arithmetic transfer operations <code>addPin</code> , <code>subPin</code> , and <code>blend</code> .
<code>rgbHiliteColor</code>	The highlight color for this graphics port.
<code>pmFgColor</code>	A handle to the palette that contains the foreground color.
<code>pmFgIndex</code>	The index value into the palette for the foreground color.
<code>pmBkColor</code>	A handle to the palette that contains the background color.
<code>pmBkIndex</code>	The index value into the palette for the background color.
<code>pmFlags</code>	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 8-3.

Table 8-3 Initial values in the `CGrafPort` record

Field	Data type	Initial setting
<code>device</code>	Integer	0 (the screen)
<code>portPixMap</code>	<code>PixMapHandle</code>	Handle to the port’s <code>PixMap</code> record
<code>portVersion</code>	Integer	\$C000
<code>grafVars</code>	Handle	Handle to a <code>GrafVars</code> record where black is assigned to the <code>rgbOpColor</code> field, the default highlight color is assigned to the <code>rgbHiliteColor</code> field, and all other fields are set to 0
<code>chExtra</code>	Integer	0
<code>pnLocHFrac</code>	Integer	The value in this field represents the low word of a <code>Fixed</code> number; in decimal, its initial value is 0.5.
<code>portRect</code>	<code>Rect</code>	<code>screenBits.bounds</code> (boundary for entire main screen)
<code>visRgn</code>	<code>RgnHandle</code>	Handle to a rectangular region coincident with <code>screenBits.bounds</code>

Table 8-3 Initial values in the `CGrafPort` record (continued)

Field	Data type	Initial setting
<code>clipRgn</code>	<code>RgnHandle</code>	Handle to the rectangular region $(-32768, -32768, 32767, 32767)$
<code>bkPixPat</code>	<code>Pattern</code>	White
<code>rgbFgColor</code>	<code>RGBColor</code>	Black
<code>rgbBkColor</code>	<code>RGBColor</code>	White
<code>pnLoc</code>	<code>Point</code>	(0,0)
<code>pnSize</code>	<code>Point</code>	(1,1)
<code>pnMode</code>	<code>Integer</code>	<code>patCopy</code>
<code>pnPixPat</code>	<code>PixPatHandle</code>	Black
<code>fillPixPat</code>	<code>PixPatHandle</code>	Black
<code>pnVis</code>	<code>Integer</code>	0 (visible)
<code>txFont</code>	<code>Integer</code>	0 (system font)
<code>txFace</code>	<code>Style</code>	Plain
<code>txMode</code>	<code>Integer</code>	<code>srcOr</code>
<code>txSize</code>	<code>Integer</code>	0 (system font size)
<code>spExtra</code>	<code>Fixed</code>	0
<code>fgColor</code>	<code>LongInt</code>	<code>blackColor</code>
<code>bkColor</code>	<code>LongInt</code>	<code>whiteColor</code>
<code>colrBit</code>	<code>Integer</code>	0
<code>patStretch</code>	<code>Integer</code>	0
<code>picSave</code>	<code>Handle</code>	NIL
<code>rgnSave</code>	<code>Handle</code>	NIL
<code>polySave</code>	<code>Handle</code>	NIL
<code>grafProcs</code>	<code>CQDProcsPtr</code>	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 8-3 on page 9-634.

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 9-628. To define your own pixel pattern, you can create a pixel pattern resource, which is described on page 9-673, or you can use the `NewPixPat` function, which is described on page 9-658.

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 9-628. To define your own pixel pattern, you can create a pixel pattern resource, which is described on page 9-673, or you can use the `NewPixPat` function, which is described on page 9-658.

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 `FillRect`, `FillRoundRect`, `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 9-596, 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 8-4 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 8-4 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 `FillRect`, `FillOval`, `FillRoundRect`, `FillArc`, `FillRgn`, 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 9-649.

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 8-4 on page 9-641 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 9-650.

SetCPixel

To set the color of an individual pixel, use the `SetCPixel` procedure.

```
PROCEDURE SetCPixel (h,v: Integer; cPix: RGBColor);
```

<code>h</code>	The horizontal coordinate of the point at the upper-left corner of the pixel.
<code>v</code>	The vertical coordinate of the point at the upper-left corner of the pixel.
<code>cPix</code>	An <code>RGBColor</code> 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 9-650.

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);
```

<code>r</code>	The rectangle that defines the oval's boundaries.
<code>startAngle</code>	The angle indicating the start of the arc.
<code>arcAngle</code>	The angle indicating the arc's extent.
<code>ppat</code>	A handle to the <code>PixPat</code> 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);
```

<code>poly</code>	A handle to the polygon to be filled.
<code>ppat</code>	A handle to the <code>PixPat</code> 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 9-608.

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 9-611.

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 8-4 on page 9-641. 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 9-640, 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 8-4 on page 9-641.

SPECIAL CONSIDERATIONS

This procedure is available for basic QuickDraw only in System 7.

SEE ALSO

You can use the `RGBBackColor` procedure, described on page 9-642, 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 9-643, 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);
```

<code>srcBits</code>	The source image. If the image is in a pixel map, you must coerce its <code>PixMap</code> record to a <code>BitMap</code> record.
<code>dstBits</code>	The destination mask.
<code>srcRect</code>	The rectangle of the source image.
<code>dstRect</code>	The rectangle of the destination image.
<code>seedH</code>	The horizontal position of the seed point.
<code>seedV</code>	The vertical position of the seed point.
<code>matchProc</code>	An optional color search function.
<code>matchData</code>	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 9-671 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);
```

<code>srcBits</code>	The source image. If the image is in a pixel map, you must coerce its <code>PixMap</code> record to a <code>BitMap</code> record.
<code>dstBits</code>	The destination image, a <code>BitMap</code> record.
<code>srcRect</code>	The rectangle of the source image.
<code>dstRect</code>	The rectangle of the destination image.
<code>seedRGB</code>	An <code>RGBColor</code> record specifying the color for pixels that should not be filled.
<code>matchProc</code>	An optional matching procedure.
<code>matchData</code>	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 9-671 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 9-662) to read such a table from a resource file; you can then use the `DisposeCTable` procedure (described on page 9-663) 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 9-673).

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 9-661).

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 9-673. 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 9-628) and returns a handle to it. This function calls the `NewPixMap` function to allocate the pattern's `PixMap` record (described on page 9-616) 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 `patlData` 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

$(\text{width of bounds}) \times \text{pixelSize} / 8$

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, $([\text{width of bounds}] \times \text{pixelSize})$ 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 9-661, 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 9-628.

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 9-667.

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 8-5.

Table 8-5 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 8-6. These default color tables are illustrated in Plate 1 at the front of this book.

Table 8-6 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 9-674.

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.

SPECIAL CONSIDERATIONS

The QDError function does not report errors returned by basic QuickDraw.

SEE ALSO

Listing 11-8 on page 12-769 in the chapter “Pictures” in this book illustrates 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 9-630.

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 9-626), 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
<code>_QDExtensions</code>	<code>\$00040007</code>

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 9-628), 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
<code>_QDExtensions</code>	<code>\$00040008</code>

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 9-618), 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.

Color QuickDraw

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. ♦

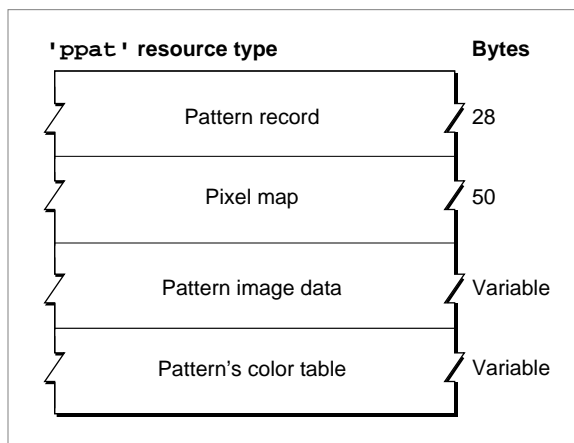
The Pixel Pattern Resource

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 9-658) 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 9-628.)

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 8-16.

Figure 8-16 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 9-628), 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 9-616), 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.

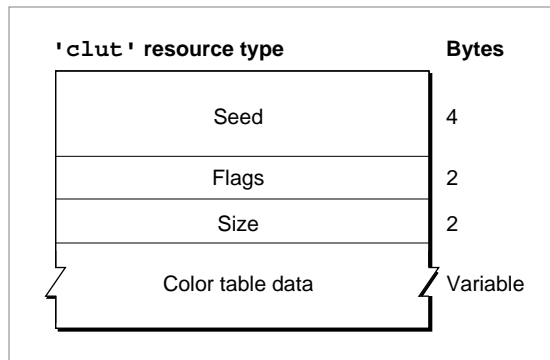
The Color Table Resource

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 9-662. 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 8-17.

Figure 8-17 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.

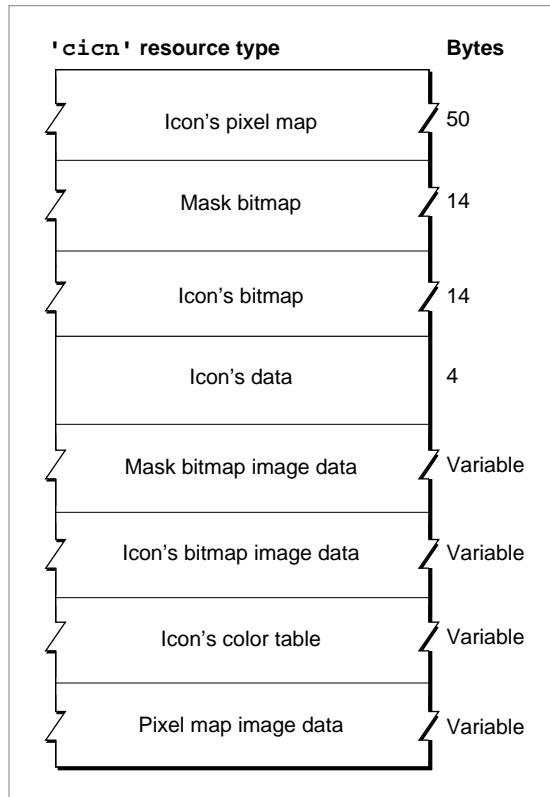
Using color icon resources, you can create icons similar to the ones that the Finder uses to display your application's files on the desktop; however, the Finder does *not* use or display any resources that you create of type 'cicn'. Instead, your application uses icon resources of type 'cicn' to display 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 8-18.

Figure 8-18 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.

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

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

Graphics Devices

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.

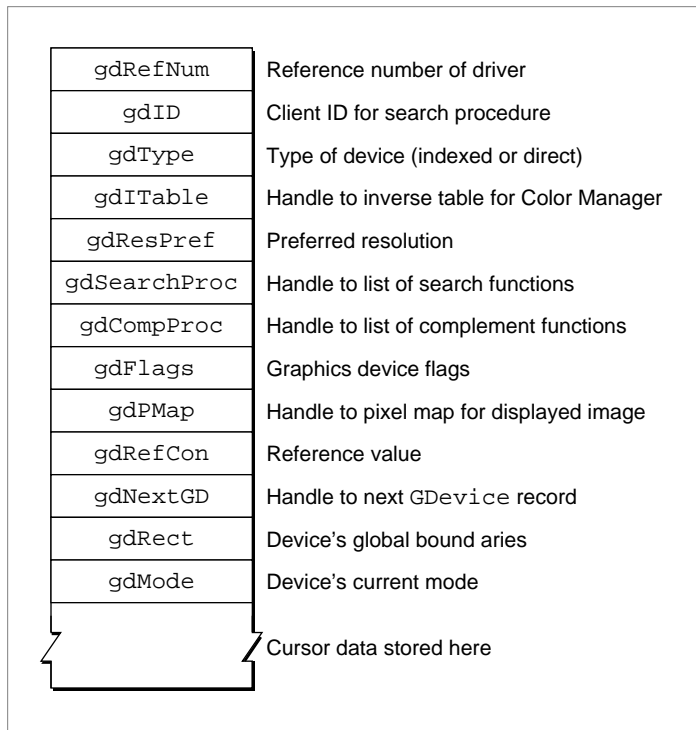
When the user moves a window or creates a window on another screen, and your application draws into that window, QuickDraw automatically makes the video device for that screen the current device. Color QuickDraw stores that information in the global variable `TheGDevice`. As Color QuickDraw draws across 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 9-1. Some aspects of its contents are discussed after the figure; see page 10-691 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 9-1 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 10-705, 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 9-1.

Listing 9-1 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;
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 9-1, 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 9-2.

Listing 9-2 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 9-3 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 9-3 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}
```

Graphics Devices

```

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.}
        END
    END
END

```

Graphics Devices

```

    {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 9-3 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 9-3 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 9-3. ▲

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 10-710, and the `HasDepth` function is described on page 10-709.

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 `PixelFormat` record. Instead, use the `NewPixelFormat` function or the `CopyPixelFormat` procedure, and fill all the fields. (These routines are described in the chapter "Color QuickDraw.") The `NewPixelFormat` function returns a `PixelFormat` record that is cloned from the `PixelFormat` record pointed to by the global variable `TheGDevice`. If you don't want a copy of the main screen's `PixelFormat` 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 `PixelFormat` record. For future compatibility you should also set the `packType`, `packSize`, `planeBytes`, and `pmReserved` fields to 0. Don't assume a `PixelFormat` record has a color table—a pixel map for a direct device doesn't need one. For compatibility, a `PixelFormat` 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 }
```

Graphics Devices

```

                                { 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 <i>Inside Macintosh: Advanced Color Imaging</i> .
gdResPref	The preferred resolution for inverse tables.
gdSearchProc	A handle to the list of search functions, as described in the chapter “Color Manager” in <i>Inside Macintosh: Advanced Color Imaging</i> ; 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 <i>Inside Macintosh: Advanced Color Imaging</i> ; 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 10-698)—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 <i>Designing Cards and Drivers for the Macintosh Family</i> , 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 10-705), 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
    (singleDevices,      {for flags parameter of DeviceLoop}
      {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

<code>singleDevices</code>	If this flag is not set, <code>DeviceLoop</code> 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 <code>singleDevices</code> flag, then <code>DeviceLoop</code> 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.
<code>dontMatchSeeds</code>	If you set the <code>dontMatchSeeds</code> flag, then <code>DeviceLoop</code> doesn't consider color table seeds when comparing graphics devices for similarity; <code>DeviceLoop</code> ignores this flag if you set the <code>singleDevices</code> flag. Used primarily by the Palette Manager, the <code>ctSeed</code> field of a <code>ColorTable</code> record is described in the chapter “Color QuickDraw” in this book.
<code>allDevices</code>	If you set the <code>allDevices</code> flag, <code>DeviceLoop</code> ignores the <code>drawingRgn</code> parameter and calls your drawing procedure for every graphics device. The value of current graphics port's <code>visRgn</code> 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;
```

<code>refNum</code>	Reference number of the graphics device for which you are creating a <code>GDevice</code> record. For most video devices, this information is set at system startup.
<code>mode</code>	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 10-698, 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 10-691. 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);
```

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

Graphics Devices

`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 10-691. 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);
```

Graphics Devices

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 9-3 on page 10-686 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 9-3 on page 10-686 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 9-3 on page 10-686 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 10-709. To change the pixel depth of a video device, your application can use the `SetDepth` function, described on page 10-710.

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:

<code>singleDevices</code>	If this flag is not set, <code>DeviceLoop</code> 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 <code>singleDevices</code> flag, then <code>DeviceLoop</code> 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.
<code>dontMatchSeeds</code>	If you set the <code>dontMatchSeeds</code> flag, then <code>DeviceLoop</code> doesn't consider the <code>ctSeed</code> field of <code>ColorTable</code> records for graphics devices when comparing them; <code>DeviceLoop</code> ignores this flag if you set the <code>singleDevices</code> flag.
<code>allDevices</code>	If you set the <code>allDevices</code> flag, <code>DeviceLoop</code> ignores the <code>drawingRgn</code> parameter and calls your drawing procedure for every device. The value of current graphics port's <code>visRgn</code> 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 9-1 on page 10-684 illustrates the use of `DeviceLoop`. See page 10-711 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 9-3 on page 10-686 illustrates the use of `TestDeviceAttribute`. Your application can use the `SetDeviceAttribute` procedure, described on page 10-698, 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 10-691). The horizontal and vertical resolutions for a graphics device are stored in the `hRes` and `vRes` fields, respectively, of the `PixelFormat` 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;
```

<code>aDevice</code>	A handle to the <code>GDevice</code> record of the video device whose pixel depth you wish to change.
<code>depth</code>	The mode ID returned by the <code>HasDepth</code> 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 <code>HasDepth</code> function to ensure that the device supports this depth.
<code>whichFlags</code>	The <code>gdDevType</code> constant, which represents a bit in the <code>gdFlags</code> field of the <code>GDevice</code> record. (If this bit is set to 0 in the <code>GDevice</code> record, the video device is black and white; if the bit is set to 1, the device supports color.)
<code>flags</code>	The value 0 or 1. If you pass 0 in this parameter, the <code>SetDepth</code> function changes the video device to black and white; if you pass 1 in this parameter, <code>SetDepth</code> 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 10-705) 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 10-691) for the current device:

```
CONST {flag bits for gdFlags field of GDevice record}
gdDevType      = 0;  {if bit is set to 1, graphics }
                  { devicesupports 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 10-691) 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 9-2 on page 10-685 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.

Offscreen Graphics Worlds

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 Worlds

offscreen graphics world. Listing 10-1 illustrates how to create an offscreen graphics world.

Listing 10-1 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
```

Offscreen Graphics Worlds

```

; {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 10-1, 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 10-1 on page 11-718, `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 10-1 on page 11-718 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 `PixelFormat` 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 `GetPixelFormat` function (described on page 11-750) to obtain a pointer to the `PixelFormat` 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 10-1 on page 11-718.

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 10-1.

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 11-735, 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 10-2 shows the code that produces the window shown in Plate 3.

Listing 10-2 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}
```

Offscreen Graphics Worlds

```

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}

```

Offscreen Graphics Worlds

```

    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 <code>flags</code> parameter of the <code>NewGWorld</code> function, <code>NewGWorld</code> (described on page 11-728) makes the base address for the offscreen pixel image purgeable.
noNewDevice	If you specify this flag for the <code>flags</code> parameter of the <code>NewGWorld</code> function, <code>NewGWorld</code> does not create a new offscreen <code>GDevice</code> record; instead, <code>NewGWorld</code> uses either the <code>GDevice</code> record you specify or the <code>GDevice</code> record for a video card on the user's system.
useTempMem	If you specify this in the <code>flags</code> parameter of the <code>NewGWorld</code> function, <code>NewGWorld</code> 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 <code>AllowPurgePixels</code> procedure (described on page 11-746) so that other applications can launch.
keepLocal	If you specify this in the <code>flags</code> parameter of the <code>NewGWorld</code> function, <code>NewGWorld</code> 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 <code>state</code> parameter of the <code>SetPixelsState</code> procedure (described on page 11-749), <code>SetPixelsState</code> makes the base address for an offscreen pixel map purgeable. If you use the <code>SetPixelsState</code> procedure without passing it this flag, then <code>SetPixelsState</code> makes the base address for an offscreen pixel map unpurgeable. If the <code>GetPixelsState</code> function (described on page 11-748) returns this flag, then the base address for an offscreen pixel is purgeable.

<code>pixelsLocked</code>	If you specify this flag for the <code>state</code> parameter of the <code>SetPixelsState</code> procedure, <code>SetPixelsState</code> locks the base address for an offscreen pixel image. If you use the <code>SetPixelsState</code> procedure without passing it this flag, then <code>SetPixelsState</code> unlocks the offscreen pixel image. If the <code>GetPixelsState</code> function returns this flag, then the base address for an offscreen pixel is locked.
<code>gWorldFlag8</code>	Reserved.
<code>gWorldFlag9</code>	Reserved.
<code>gWorldFlag10</code>	Reserved.
<code>gWorldFlag11</code>	Reserved.
<code>gWorldFlag12</code>	Reserved.
<code>gWorldFlag13</code>	Reserved.
<code>gWorldFlag14</code>	Reserved.
<code>gWorldFlag15</code>	Reserved.
<code>mapPix</code>	If the <code>UpdateGWorld</code> function (described on page 11-735) returns this flag, then it remapped the colors in the offscreen pixel map to a new color table.
<code>newDepth</code>	If the <code>UpdateGWorld</code> function returns this flag, then it translated the offscreen pixel map to a different pixel depth.
<code>alignPix</code>	If the <code>UpdateGWorld</code> function returns this flag, then it realigned the offscreen pixel image to an onscreen window.
<code>newRowBytes</code>	If the <code>UpdateGWorld</code> function returns this flag, then it changed the <code>rowBytes</code> field of the <code>PixMap</code> record for the offscreen graphics world.
<code>reallocPix</code>	If the <code>UpdateGWorld</code> 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.
<code>gWorldFlag21</code>	Reserved.
<code>gWorldFlag22</code>	Reserved.
<code>gWorldFlag23</code>	Reserved.
<code>gWorldFlag24</code>	Reserved.
<code>gWorldFlag25</code>	Reserved.
<code>gWorldFlag26</code>	Reserved.
<code>gWorldFlag27</code>	Reserved.
<code>clipPix</code>	If the <code>UpdateGWorld</code> function returns this flag, then it clipped the pixel image.
<code>stretchPix</code>	If the <code>UpdateGWorld</code> function returns this flag, then it stretched or shrank the offscreen image.
<code>ditherPix</code>	If the <code>UpdateGWorld</code> function returns this flag, then it dithered the offscreen image.
<code>gwFlagErr</code>	If the <code>UpdateGWorld</code> 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 11-740) 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 11-744. ♦

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;
```


Offscreen Graphics Worlds

`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.

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 11-750) 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 11-744) 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

Offscreen Graphics Worlds

graphics world, and the `SetGWorld` procedure (described on page 11-740) 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 11-746).
- 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
<code>_QDExtensions</code>	<code>\$00160000</code>

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Illegal parameter
<code>cDepthErr</code>	-157	Invalid pixel depth

SEE ALSO

Listing 10-1 on page 11-718 and Listing 10-2 on page 11-722 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 11-735.

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 un purgeable.

`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
<code>_QDExtensions</code>	<code>\$000E0010</code>

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Illegal parameter
<code>cNoMemErr</code>	-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;
```

<code>globalRect</code>	The boundary rectangle, in global coordinates, for the offscreen pixel map.
<code>purgeable</code>	A value of <code>TRUE</code> to make the memory block for the offscreen pixel map purgeable, or a value of <code>FALSE</code> to make it un purgeable.
<code>gdh</code>	The handle to the <code>GDevice</code> record for the graphics device with the greatest pixel depth among all graphics devices whose boundary rectangles intersect the rectangle specified in the <code>globalRect</code> parameter.
<code>offscreenPixMap</code>	A handle to the new offscreen <code>PixMap</code> 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
<code>_QDExtensions</code>	<code>\$000E0015</code>

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.

Offscreen Graphics Worlds

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 11-744) 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
<code>mapPix</code>	Color QuickDraw remapped the colors to a new color table.
<code>newDepth</code>	Color QuickDraw translated the pixel values in the offscreen pixel image to those for a different pixel depth.
<code>alignPix</code>	QuickDraw realigned the offscreen image to the window.
<code>newRowBytes</code>	QuickDraw changed the value of the <code>rowBytes</code> field in the <code>PixelFormat</code> record for the offscreen graphics world.
<code>reallocPix</code>	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.
<code>clipPix</code>	QuickDraw clipped the pixel image.
<code>stretchPix</code>	QuickDraw stretched or shrank the offscreen image.
<code>ditherPix</code>	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
<code>_QDExtensions</code>	<code>\$00160003</code>

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
<code>_QDExtensions</code>	<code>\$00040004</code>

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
<code>_QDExtensions</code>	<code>\$00040011</code>

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
<code>_QDExtensions</code>	<code>\$00080005</code>

SEE ALSO

Listing 10-1 on page 11-718 and Listing 10-2 on page 11-722 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);
```

<code>port</code>	A pointer to an offscreen graphics world, color graphics port, or basic graphics port.
<code>gdh</code>	A handle to a <code>GDevice</code> record. If you pass a pointer to an offscreen graphics world in the <code>port</code> parameter, set this parameter to <code>NIL</code> , because <code>SetGWorld</code> 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
<code>_QDExtensions</code>	<code>\$00080006</code>

SEE ALSO

Listing 10-1 on page 11-718 and Listing 10-2 on page 11-722 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

Offscreen Graphics Worlds

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
<code>_QDExtensions</code>	<code>\$00040012</code>

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
<code>_QDExtensions</code>	<code>\$00040017</code>

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 11-743.

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 11-735) 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
<code>_QDExtensions</code>	<code>\$00040001</code>

SEE ALSO

Listing 10-1 on page 11-718 and Listing 10-2 on page 11-722 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
<code>_QDExtensions</code>	<code>\$00040002</code>

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 un purgeable base address for an offscreen pixel image.

To get a handle to an offscreen pixel map, first use the `GetGWorldPixMap` function, described on page 11-743. Then supply this handle for the `pm` parameter of `AllowPurgePixels`.

Your application should call the `LockPixels` function (described on page 11-744) 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 11-735) 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
<code>_QDExtensions</code>	<code>\$0004000B</code>

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 un purgeable. To get a handle to an offscreen pixel map, use the `GetGWorldPixMap` function, described on page 11-743. 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
<code>_QDExtensions</code>	<code>\$0004000C</code>

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 un purgeable. 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 11-743.

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
<code>_QDExtensions</code>	<code>\$0004000D</code>

SEE ALSO

After using `GetPixelsState` and before using `SetPixelsState`, your application can temporarily use the `AllowPurgePixels` procedure (described on page 11-746) to make the base address for an offscreen pixel image purgeable, the `NoPurgePixels` procedure (described on page 11-747) to make it unpurgeable, the `LockPixels` function (described on page 11-744) to prevent it from being moved, and the `UnlockPixels` procedure (described on page 11-745) 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 11-745 and page 11-746, 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
<code>_QDExtensions</code>	<code>\$0008000E</code>

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 11-746) to temporarily mark the memory block for its offscreen pixel map as purgeable, the `NoPurgePixels` procedure (described on page 11-747) to make it un purgeable, the `LockPixels` function (described on page 11-744) to prevent it from being moved, and the `UnlockPixels` procedure (described on page 11-745) to unlock it.

GetPixBaseAddr

You can use the `GetPixBaseAddr` function to obtain a pointer to an offscreen pixel map.

```
FUNCTION GetPixBaseAddr (pm: PixMapHandle): Ptr;
```

<code>pm</code>	A handle to an offscreen pixel map. To get a handle to an offscreen pixel map, use the <code>GetGWorldPixMap</code> function, described on page 11-743.
-----------------	---

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
<code>_QDExtensions</code>	<code>\$0004000F</code>

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 11-743. 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
<code>_QDExtensions</code>	<code>\$00040016</code>

SEE ALSO

See *Inside Macintosh: Memory* for information about determining and setting addressing modes.

Result Codes

noErr	0	No error
paramErr	-50	Illegal parameter
cNoMemErr	-152	Failed to allocate memory for structures
cDepthErr	-157	Invalid pixel depth

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 `OpenPicture` 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

Pictures

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 11-1.

Figure 11-1 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. ♦

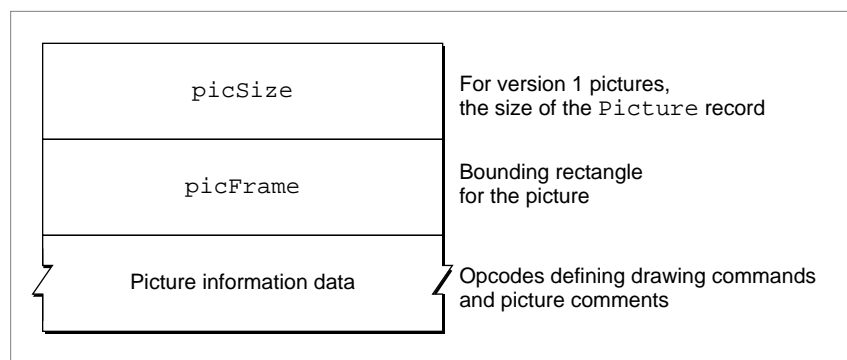
Picture Formats

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 11-2, the `Picture` record begins with a `picSize` field and a `picFrame` field, followed by a variable amount of picture definition data.

Figure 11-2 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

Pictures

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

Pictures

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 `OpenPicture` 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

Pictures

- 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

Pictures

- 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`. ♦

Pictures

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 11-1 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 11-1 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}

```

Pictures

```

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 11-1, always sets a valid clipping region. ▲

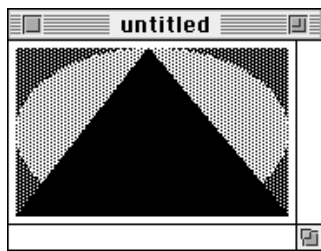
The `MyCreateAndDrawPict` routine uses `QuickDraw` commands to draw a filled rectangle, a filled oval, and a black triangle. These commands are stored in the `Picture` record.

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 11-3 shows the resulting figure.

Figure 11-3 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 11-2 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 11-2 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}
```

Pictures

```

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 11-7 on
page 12-768}
    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 11-2, 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 `FSOpenDF` 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 11-3 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 11-10 on page 12-772 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 11-3 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 11-5}
        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 11-5}
        currPort^.grafProcs := @customProcs; {replace current QDProcs record}
    END;
    MyReplaceGetPic := savedProcs;
END;

```

Pictures

Listing 11-4 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 11-4 Determining whether a graphics port is color or basic

```
FUNCTION MyIsColorPort(aPort: GrafPtr): Boolean;
BEGIN
    MyIsColorPort := (aPort^.portBits.rowBytes < 0)
END;
```

Listing 11-5 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 11-5 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 11-6 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 11-6 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 11-7}
        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;
END;
```

Pictures

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 11-7 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 11-7 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 12-796) to gather information about a picture. The `PictInfo` record (described on page 12-781) 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 11-8 illustrates an application-defined routine, called `MyDrawResPICT`, that retrieves and draws a picture stored as a resource.

Listing 11-8 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 11-7 on page 12-768}
        DrawPicture(myPic, destRect);
```

Pictures

```

    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`, `FSPWrite`, and `FSPClose`. The use of these routines is illustrated in Listing 11-9, 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 11-9, your application should store the data (that is, the Picture record) after this 512-byte header.

Listing 11-9 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;

```

Pictures

```

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 11-10 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 11-3 on page 12-765 shows how to replace QuickDraw's standard `StdGetPic` procedure with one that reads data from a spool file.)

Pictures

Listing 11-10 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 11-4 on page 12-766}
    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 11-11}
        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 11-11 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 11-11 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;
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 12-781). 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 11-12 shows an application-defined routine, called `MyGetPICTProfileCount`, that uses `GetPictInfo` to record comments in a `CommentSpec` record (which is described on page 12-779). The `MyGetPICTProfileCount` routine uses the `CommentSpec` record to determine whether any color profiles are included in the picture as picture comments.

Pictures

Listing 11-12 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;
        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 12-810.

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.

Pictures

“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 12-786 and page 12-788, respectively), `QuickDraw` begins collecting your subsequent drawing commands in a `Picture` record. (You use the `ClosePicture` procedure, described on page 12-791, to complete a picture definition.) When you use the `GetPicture` function (described on page 12-795) to retrieve a picture stored in a resource, `GetPicture` reads the resource into memory as a `Picture` record. (`'PICT'` resources are described on page 12-816.) By using the `DrawPicture` procedure (described on page 12-793), you can draw onscreen the picture defined by the commands stored in the `Picture` record.

Pictures

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

<code>picSize</code>	The size of the rest of this record for a version 1 picture. To maintain compatibility with the version 1 picture format, the <code>picSize</code> 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 <code>picSize</code> field, you should use the Memory Manager function <code>GetHandleSize</code> to determine the size of a picture in memory, the File Manager function <code>PBGetFInfo</code> to determine the size of a picture in a 'PICT' file, and the Resource Manager function <code>MaxSizeResource</code> to determine the size of a 'PICT' resource. (See <i>Inside Macintosh: Memory</i> , <i>Inside Macintosh: Files</i> , and <i>Inside Macintosh: More Macintosh Toolbox</i> for more information about these functions.)
<code>picFrame</code>	The bounding rectangle for the picture defined in the rest of this record. The <code>DrawPicture</code> 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 12-789, 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 12-786) 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

<code>srcRect</code>	The optimal bounding rectangle for the resolution indicated by the next two fields. When you later call the <code>DrawPicture</code> procedure (described on page 12-793) to play back the saved picture, you supply a destination rectangle, and <code>DrawPicture</code> 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
<code>hRes</code>	The best horizontal resolution for the picture. Notice that this value is of type <code>Fixed</code> —a value of \$00480000 specifies a horizontal resolution of 72 dpi.
<code>vRes</code>	The best vertical resolution for the picture. Notice that this value is of type <code>Fixed</code> —a value of \$00480000 specifies a horizontal resolution of 72 dpi.
<code>version</code>	Always set this field to -2.
<code>reserved1</code>	Reserved; set to 0.
<code>reserved2</code>	Reserved; set to 0.

CommentSpec

If you specify the `recordComments` constant in the `verb` parameter to the `GetPictInfo` function (described on page 12-796) or the `NewPictInfo` function (described on page 12-802), you receive a `PictInfo` record (described beginning on page 12-781) 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 <code>GetPictInfo</code> function or in all the pictures examined with the <code>NewPictInfo</code> function.
ID	The value set in the <code>kind</code> parameter when the picture comment was created with the <code>PicComment</code> procedure. The <code>PicComment</code> procedure is described on page 12-789. 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 11-12 on page 12-774 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 12-796) or the `NewPictInfo` function (described on page 12-802), your application receives a `PictInfo` record (described beginning on page 12-781) 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.)

Pictures

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

<code>pictFontID</code>	The ID number of the font as it is stored in the picture.
<code>sysFontID</code>	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.
<code>size</code>	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.
<code>style</code>	The styles for this font family at any of its sizes. The values in this field can also be represented with the <code>Style</code> data type:

```

TYPE
    StyleItem = (bold, italic, underline, outline,
                shadow, condense, extend);
    Style = SET OF StyleItem;
```

<code>nameOffset</code>	The offset into the list of font names (indicated by the <code>fontNamesHandle</code> field of the <code>PictInfo</code> 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 12-796) to collect information about a picture, or when you use the `GetPixMapInfo` function (described on page 12-799) 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 12-807) 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)}
```

Pictures

```

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 <code>GetPictInfo</code> function, or the number of colors in the pixel map or bitmap specified to the <code>GetPixMapInfo</code> function, or the total number of colors for all the pictures, pixel maps, and bitmaps returned by the <code>RetrievePictInfo</code> 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 12-810 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 <code>Palette</code> record if you specified to the <code>GetPictInfo</code> , <code>GetPixMapInfo</code> , or <code>NewPictInfo</code> function that colors be returned in a <code>Palette</code> record. That <code>Palette</code> 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 <code>verb</code> parameter to the function, the <code>Palette</code> 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 <code>NIL</code> . See the chapter "Palette Manager" in <i>Inside Macintosh: Advanced Color Imaging</i> for more information about <code>Palette</code> records.

Pictures

<code>theColorTable</code>	<p>A handle to the resulting <code>ColorTable</code> record if you specified to the <code>GetPictInfo</code>, <code>GetPixMapInfo</code>, or <code>NewPictInfo</code> function that colors be returned in a <code>ColorTable</code> record. If the pictures, pixel maps, or bitmaps contain fewer colors found than you specified to the function, the unused entries in the <code>ColorTable</code> record are filled with black. Depending on the constant you pass in the <code>verb</code> parameter to the function, the <code>ColorTable</code> 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 <code>ColorTable</code> records. On Macintosh computers running basic QuickDraw only, this field is always returned as <code>NIL</code>.</p> <p>If a picture has more than 256 colors or has pixel depths of 32 bits, then Color QuickDraw translates the colors in the <code>ColorTable</code> record to 16-bit depths. In such a case, the returned colors might have a slight loss of resolution, and the <code>uniqueColors</code> field reflects the number of colors distinguishable at that pixel depth.</p>
<code>hRes</code>	The horizontal resolution of the current picture, pixel map, or bitmap retrieved by the <code>GetPictInfo</code> or <code>GetPixMapInfo</code> function; the greatest horizontal resolution from all pictures, pixel maps, and bitmaps retrieved by the <code>RetrievePictInfo</code> function.
<code>vRes</code>	The vertical resolution of the current picture, pixel map, or bitmap retrieved by the <code>GetPictInfo</code> or <code>GetPixMapInfo</code> function; the greatest vertical resolution of all pictures, pixel maps, and bitmaps retrieved by the <code>RetrievePictInfo</code> function. Note that although the values of the <code>hRes</code> and <code>vRes</code> fields are usually the same, they don't have to be.
<code>depth</code>	The pixel depth of the picture specified to the <code>GetPictInfo</code> function or the pixel map specified to the <code>GetPixMapInfo</code> function. When you use the <code>RetrievePictInfo</code> function, this field contains the deepest pixel depth of all pictures or pixel maps retrieved by the function.
<code>sourceRect</code>	The optimal bounding rectangle for displaying the picture at the resolution indicated by the <code>hRes</code> and <code>vRes</code> fields. The upper-left corner of the rectangle is always (0,0). Pictures created with the <code>OpenCPicture</code> function have the <code>hRes</code> , <code>vRes</code> , and <code>sourceRect</code> fields built into their <code>Picture</code> records. For pictures created by <code>OpenPicture</code> , the <code>hRes</code> and <code>vRes</code> fields are set to 72 dpi, and the source rectangle is calculated using the <code>picFrame</code> field of the <code>Picture</code> record for the picture.
<code>textCount</code>	The number of text strings in the picture specified to the <code>GetPictInfo</code> function, or the total number of text objects in all the pictures retrieved by the <code>RetrievePictInfo</code> function. For pixel maps and bitmaps specified to <code>GetPixMapInfo</code> or <code>RetrievePictInfo</code> , this field is set to 0.
<code>lineCount</code>	The number of lines in the picture specified to the <code>GetPictInfo</code> function, or the total number of lines in all the pictures retrieved by the <code>RetrievePictInfo</code> function. For pixel maps and bitmaps, this field is set to 0.

Pictures

<code>rectCount</code>	The number of rectangles in the picture specified to the <code>GetPictInfo</code> function, or the total number of rectangles in all the pictures retrieved by the <code>RetrievePictInfo</code> function. For pixel maps and bitmaps, this field is set to 0.
<code>rRectCount</code>	The number of rounded rectangles in the picture specified to the <code>GetPictInfo</code> function, or the total number of rounded rectangles in all the pictures retrieved by the <code>RetrievePictInfo</code> function. For pixel maps and bitmaps, this field is set to 0.
<code>ovalCount</code>	The number of ovals in the picture specified to the <code>GetPictInfo</code> function, or the total number of ovals in all the pictures retrieved by the <code>RetrievePictInfo</code> function. For pixel maps and bitmaps, this field is set to 0.
<code>arcCount</code>	The number of arcs and wedges in the picture specified to the <code>GetPictInfo</code> function, or the total number of arcs and wedges in all the pictures retrieved by the <code>RetrievePictInfo</code> function. For pixel maps and bitmaps, this field is set to 0.
<code>polyCount</code>	The number of polygons in the picture specified to the <code>GetPictInfo</code> function, or the total number of polygons in all the pictures retrieved by the <code>RetrievePictInfo</code> function. For pixel maps and bitmaps, this field is set to 0.
<code>regionCount</code>	The number of regions in the picture specified to the <code>GetPictInfo</code> function, or the total number of regions in all the pictures retrieved by the <code>RetrievePictInfo</code> function. For pixel maps and bitmaps, this field is set to 0.
<code>bitMapCount</code>	The total number of bitmaps in the survey.
<code>pixMapCount</code>	The total number of pixel maps in the survey.
<code>commentCount</code>	The number of comments in the picture specified to the <code>GetPictInfo</code> function, or the total number of comments in all the pictures retrieved by the <code>RetrievePictInfo</code> function. This field is valid only if you specified to the <code>GetPictInfo</code> or <code>NewPictInfo</code> function that comments be returned in a <code>CommentSpec</code> record, described on page 12-779. For pixel maps and bitmaps, this field is set to 0.
<code>uniqueComments</code>	The number of picture comments that have different IDs in the picture specified to the <code>GetPictInfo</code> function, or the total number of picture comments with different IDs in all the pictures retrieved by the <code>RetrievePictInfo</code> 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 <code>CommentSpec</code> record. For pixel maps and bitmaps, this field is set to 0.
<code>commentHandle</code>	A handle to an array of <code>CommentSpec</code> records, described on page 12-779. For pixel maps and bitmaps, this field is set to <code>NIL</code> .

Pictures

<code>uniqueFonts</code>	<p>The number of different fonts in the picture specified to the <code>GetPictInfo</code> function, or the total number of different fonts in all the pictures retrieved by the <code>RetrievePictInfo</code> 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.</p> <p>This field is valid only if you specify that fonts be returned in a <code>FontSpec</code> record, which is described on page 12-779. For pixel maps and bitmaps, this field is set to 0.</p>
<code>fontHandle</code>	<p>A handle to a list of <code>FontSpec</code> records, described on page 12-779. For pixel maps and bitmaps, this field is set to <code>NIL</code>.</p>
<code>fontNamesHandle</code>	<p>A handle to the names of the fonts in the picture retrieved by the <code>GetPictInfo</code> function or the pictures retrieved by the <code>RetrievePictInfo</code> function. The offset to a particular name is stored in the <code>nameOffset</code> field of the <code>FontSpec</code> 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.</p>

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`.)

Pictures

The `OpenPicture` function works on all Macintosh computers running System 7. Pictures created with the `OpenPicture` 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.

OpenPicture

To begin defining a picture in extended version 2 format, use the `OpenPicture` function.

```
FUNCTION OpenPicture (newHeader: OpenCPicParams): PicHandle;
```

newHeader An `OpenCPicParams` record, which is defined as follows (see page 12-778 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 `OpenPicture` function returns a handle to a new `Picture` record (described on page 12-776). 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 12-789) to include picture comments in your picture definition.

The `OpenPicture` 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 12-793) 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:

$\text{scale factor} = \text{destination resolution} / \text{source resolution}$

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 12-791. To draw the picture, use the `DrawPicture` procedure, described on page 12-793.

After creating the picture, your application can use the `GetPictInfo` function (described on page 12-796) to gather information about it. The `PictInfo` record (described on page 12-781) 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

Pictures

`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 11-1 on page 12-761, always sets a valid clipping region.

The `OpenPicture` 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 11-1 on page 12-761 illustrates the use of the `OpenPicture` 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 `OpenPicture` 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 12-776). 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 12-789) 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 12-791. To draw the picture, use the `DrawPicture` procedure, described on page 12-793.

SPECIAL CONSIDERATIONS

The version 2 and version 1 picture formats support only 72-dpi resolution. The `OpenPicture` function creates pictures in the extended version 2 format. The extended version 2 format, which is created by the `OpenPicture` function on all Macintosh computers running System 7, permits your application to specify additional resolutions when creating images.

See the description of the `OpenPicture` 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);
```

<code>kind</code>	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.
<code>dataSize</code>	Size of any additional data passed in the <code>dataHandle</code> 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.
<code>dataHandle</code>	A handle to additional data, if used. If no additional data is used, specify <code>NIL</code> in this parameter.

Pictures

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 11-1 on page 12-761 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.

Pictures

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 12-786) 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 12-763. 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 11-6 on page 12-767. ♦

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 12-796) to gather information about a picture. The `PictInfo` record (described on page 12-781) 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

Pictures

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 11-1 on page 12-761, 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 11-1 on page 12-761 illustrates how to use `DrawPicture` after creating a picture while your application is running; Listing 11-2 on page 12-763 illustrates how to use `DrawPicture` after reading in a picture stored in a 'PICT' file; Listing 11-6 on page 12-767 illustrates how to use `DrawPicture` after reading in a picture stored in the scrap; and Listing 11-8 on page 12-769 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 12-793) 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 12-816.

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.

Pictures

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 12-781.

Pictures

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.

Pictures

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 12-799) 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
<code>_Pack15</code>	<code>\$0800</code>

RESULT CODES

<code>pictInfoVersionErr</code>	-11000	Version number not 0
<code>pictInfoVerbErr</code>	-11002	Invalid verb combination specified
<code>cantLoadPickMethodErr</code>	-11003	Custom pick method not in resource chain
<code>colorsRequestedErr</code>	-11004	Number out of range or greater than that passed to <code>NewPictInfo</code>
<code>pictureDataErr</code>	-11005	Invalid picture data

SEE ALSO

The `PictInfo` record is described on page 12-781, the `CommentSpec` record is described on page 12-779, and the `FontSpec` record is described on page 12-779. 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 12-810 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 11-12 on page 12-774 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.

Pictures

`thePictInfo`

A pointer to a `PictInfo` record, which will hold information about a pixel map or bitmap. The `PictInfo` record is described on page 12-781.

`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
<code>_Pack15</code>	<code>\$0801</code>

RESULT CODES

<code>pictInfoVersionErr</code>	-11000	Version number not 0
<code>pictInfoVerbErr</code>	-11002	Invalid verb combination specified
<code>cantLoadPickMethodErr</code>	-11003	Custom pick method not in resource chain
<code>colorsRequestedErr</code>	-11004	Number out of range or greater than that passed to <code>NewPictInfo</code>

SEE ALSO

See “Application-Defined Routines” beginning on page 12-810 for more information about creating your own color-picking method for the `colorPickMethod` parameter. The `PictInfo` record is described on page 12-781; 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*.

Pictures

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 12-807) to return in a `PictInfo` record (described on page 12-781). 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.

Pictures

`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 12-806. For each of these functions, you identify the survey with the ID number returned by `NewPictInfo`.

Use the `RetrievePictInfo` function (described on page 12-807) 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.

Pictures

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
<code>_Pack15</code>	<code>\$0602</code>

RESULT CODES

<code>pictInfoVersionErr</code>	-11000	Version number not 0
<code>pictInfoVerbErr</code>	-11002	Invalid verb combination specified
<code>cantLoadPickMethodErr</code>	-11003	Custom pick method not in resource chain
<code>colorsRequestedErr</code>	-11004	Number out of range or greater than that passed to <code>NewPictInfo</code>

SEE ALSO

The `PictInfo` record is described on page 12-781, the `CommentSpec` record is described on page 12-779, and the `FontSpec` record is described on page 12-779. 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 12-810 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 12-802.

`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 12-807, 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 12-799) to obtain color information about that pixel map.

The `RecordPictInfo` function may move or purge memory.

Pictures

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `RecordPictInfo` function are

Trap macro	Selector
<code>_Pack15</code>	<code>\$0403</code>

RESULT CODES

<code>pictInfoIDErr</code>	-11001	Invalid picture information ID
<code>pictureDataErr</code>	-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 12-802.

`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 12-807, 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
<code>_Pack15</code>	<code>\$0404</code>

RESULT CODES

<code>pictInfoIDErr</code>	<code>-11001</code>	Invalid picture information ID
<code>pictureDataErr</code>	<code>-11005</code>	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 12-802.

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 12-781.

colorsRequested
From 1 to 256, the number of colors you want returned in the `ColorTable` or `Palette` record included in the `PictInfo` record.

Pictures

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
<code>_Pack15</code>	<code>\$0505</code>

RESULT CODES

<code>pictInfoIDErr</code>	-11001	Invalid picture information ID
<code>colorsRequestedErr</code>	-11004	Number out of range or greater than that passed to <code>NewPictInfo</code>

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 12-802.

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 12-807. 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
<code>_Pack15</code>	<code>\$0206</code>

RESULT CODE

<code>pictInfoIDErr</code>	<code>-11001</code>	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 12-795, your application can use the `GetPictInfo`, `GetPixelFormatInfo`, 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 11-1, each selector requires your color-picking method to call a different routine, which should return its results in register D0.

Table 11-1 Routine selectors for an application-defined color-picking method

D0 value	Routine to call
0	<code>MyInitPickMethod</code>
1	<code>MyRecordColors</code>
2	<code>MyCalcColorTable</code>
3	<code>MyDisposeColorPickMethod</code>

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 12-813 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 12-814 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 12-816 as a Pascal function declared as `MyDisposeColorPickMethod`.

If your routines return an error, that error is passed back to the `GetPictInfo`, `GetPictMapInfo`, 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}
```

Pictures

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 12-814) 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.

Pictures

`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 12-811).

`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 12-811) 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 12-813) 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 12-811). 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 12-811). 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.

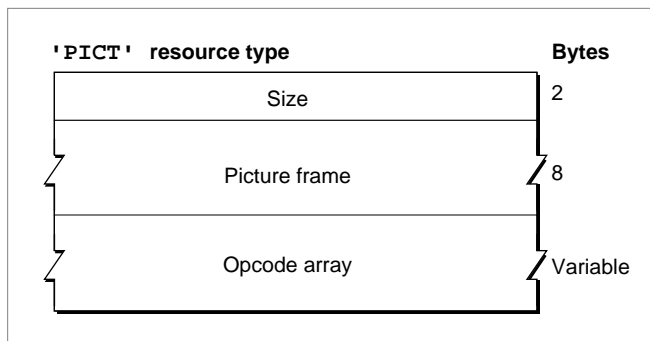
To save a picture in a 'PICT' resource while your application is running, you should use Resource Manager routines, such as FSpOpenResFile (to open your application's resource fork), ChangedResource (to change an existing 'PICT' resource), AddResource (to add a new 'PICT' resource), WriteResource (to 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 11-4, 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 11-4 Structure of a compiled picture ('PICT') resource



To retrieve a 'PICT' resource, specify its resource ID to the `GetPicture` function, described on page 12-795, which returns a handle to the picture. Listing 11-8 on page 12-769 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 12-810 for information about creating a color-picking method resource.

Result Codes

<code>pictInfoVersionErr</code>	-11000	Version number not 0
<code>pictInfoIDErr</code>	-11001	Invalid picture information ID
<code>pictInfoVerbErr</code>	-11002	Invalid verb combination specified
<code>cantLoadPickMethodErr</code>	-11003	Custom pick method not in resource chain
<code>colorsRequestedErr</code>	-11004	Number out of range or greater than that passed to <code>NewPictInfo</code>
<code>pictureDataErr</code>	-11005	Invalid picture data

Index

Symbols

& operator 266
@ operator 338

Numerals

0 (memory location) 236, 267
0-length handles 266
24-bit addressing
 defined 247
32-bit addressing
 defined 247
32-bit clean 248

A

A5 register
 and A5 world 245
 grow-zone functions saving and restoring 281
 setting and restoring 310 to 311
A5 world
 and component connections 96 to 97
 defined 244, 245
 setting 310 to 311
acceptAppDiedEvents flag 473
AcceptHighLevelEvent function 425, 429 to 431
acceptSuspendResumeEvents flag 471
access modes 135
access paths 123, 135
activation procedures 205 to 206, 234
active fields 206
addMax arithmetic transfer mode 609, 610
addOver arithmetic transfer mode 608, 610
addPin arithmetic transfer mode 608, 610, 648
adMin arithmetic transfer mode 609, 610
AESend function 443
aliases
 defined 125
 resolution by Finder 125
 resolution of by Standard File Package 189
alignPix flag 725, 727, 737
allDevices flag 706
allInit flag 693, 699, 707, 712
allocation blocks

 size 121
AllowPurgePixels procedure 746 to 747
AND operator 266
AppFile data type 149, 155
application-defined routines
 MyComponent 107
application files records 155
application global variables 244
application heap 241 to 243
 defined 242
 determining amount of free space 274 to 276
 maximizing space to prevent fragmentation 272
 setting up 270 to 274, 282 to 284, 339 to 341
application heap limit
 getting 285, 397
 setting 285 to 286, 397 to 398
application heap zone
 defined 317
 getting a pointer to 394
 initializing 400 to 401
 maximizing size of 283, 339 to 340
 .See also heap zones
 subdividing into multiple heap zones 327 to 329
application parameters 245
application partitions 236, 239 to 245
ApplicationZone function 394
ApplLimit global variable 240, 272, 285, 397
ApplZone global variable 394
arithmetic transfer modes 608 to 611, 648
auxiliary window record 492 to 493
AuxWinRec data type 492 to 493

B

background colors 642 to 643, 650
background patterns
 changing 638 to 639
 in color graphics ports 621
backing-store files 118
BackPixPat procedure 638 to 639
Balloon Help 194
basic graphics ports
 color pictures in 756 to 757
 compared with color graphics ports 575 to 579
 copying images from offscreen graphics worlds 721 to 723
 getting 720, 739 to 740

- restoring 720, 740 to 741
- saving 720, 739 to 740
- setting 720, 740 to 741
- basic QuickDraw
 - application-defined routines for 711 to 713
 - data structures in 691 to 694, 724 to 727, 776 to 778
 - resources in 713, 816 to 817
 - routines in 695 to 701, 728 to 752, 785 to 795
- BeginUpdate routine 534 to 535
- BitClr procedure 612
- bit images
 - as pixel images in offscreen graphics worlds 721
- bitmaps
 - as pixel maps in offscreen graphics worlds 716, 720 to 721
- bit patterns
 - in color graphics ports 593 to 594, 628 to 629, 638, 639, 583
- blend arithmetic transfer mode 608, 610, 648
- block contents 334
- block headers 334 to 336
- BlockMove procedure 306 to 307, 372 to 373
- blocks, memory
 - .*See also* nonrelocatable blocks; relocatable blocks
 - allocating 276 to 278
 - concatenating 377 to 379
 - copying 306 to 307, 372 to 377
 - defined 242
 - how allocated 254
 - manipulating 372 to 379
 - releasing 276 to 278
 - size correction for 335, 336
- Boolean operators, short-circuit 266
- Boolean transfer modes 602 to 608
- BringToFront routine 515 to 516
- BufPtr global variable
 - limitation on lowering during startup 398
- BufPtr global variable 326 to 327
- burstDevice flag 693, 699, 707, 712
- Button function 453
- Byte data type 337

C

- CalcCMask procedure 653 to 654
- CalcVisBehind routine 550 to 551
- CalcVis routine 550
- callback routines
 - and code segmentation 264 to 265
 - with Standard File Package routines 191, 195 to 206
- CallComponentFunction function 91 to 92
- CallComponentFunctionWithStorage function 92 to 93

- canBackground flag 471
- can do request 50
- CaptureComponent function 53, 103
- caret, time between blinks of 460
- catalog files 118
- CGrafPort 29
- CGrafPort data type
 - . *See also* color graphics ports
- CGrafPort data type 618 to 624
- CGrafPort records
 - background pattern for 621
 - clipping regions 621
 - closing 637
 - compared with GrafPort records 578 to 579
 - copying images from offscreen graphics worlds 721 to 723
 - creating 590 to 591, 633 to 636
 - disposing of 591, 633, 637
 - getting 720, 739 to 740
 - opening 633 to 636
 - pattern stretching in 623
 - pen locations in 622
 - pen modes in 622
 - pen patterns in 622
 - pen sizes in 622
 - pen visibility in 622
 - pixel maps in 620
 - port rectangles in 621
 - restoring 720, 740 to 741
 - saving 720, 739 to 740
 - setting 720, 740 to 741
 - text in 623
 - visible regions 621
- CGrafPtr 28
- CheckUpdate function 546 to 547
- 'cicn' resource type 675 to 676
- ClipAbove routine 547
- clipping regions 621
- clipPix flag 726, 727, 736, 737
- ClipRect procedure 762
- cloning
 - components 63
- Close command (File menu) 126 to 128, 146 to 148
- CloseComponent function 75
- CloseComponentResFile function 100 to 101
- CloseCPort procedure 637
- ClosePicture procedure 761, 791
- close request 49
- CloseWindow procedure 770
- CloseWindow routine 532 to 533
- closing files 146 to 148, 159 to 160
- closing windows
 - routines for 531 to 534
- ClrAppFiles procedure 174
- clumps

- defined 122
- 'clut' resource type 674 to 675
- 'cmapt' resource type 817
- code resources, copying into system heap 325
- code segmenting
 - and dangling pointers 263 to 264
 - effect on callback routines 264 to 265
- color banks 782, 810 to 811, 811 to 815
- Color control panel 475
- color graphics ports
 - background pattern for 621
 - clipping regions 621
 - closing 637
 - compared with basic graphic ports 575 to 579
 - copying images between 596 to 602
 - copying images from offscreen graphics worlds 721 to 723
 - creating 590 to 591, 633 to 636
 - data type for 618 to 624
 - disposing of 591, 633, 637
 - getting 720, 739 to 740
 - opening 633 to 636
 - pattern stretching in 623
 - pen locations in 622
 - pen modes in 622
 - pen patterns in 622
 - pen sizes in 622
 - pen visibility in 622
 - pixel maps in 620
 - port rectangles in 621
 - restoring 720, 740 to 741
 - saving 720, 739 to 740
 - setting 720, 740 to 741
 - text in 623
 - visible regions 621
- color icon resources 675 to 676
- color-picking method resources 817
- Color QuickDraw
 - and color window records 484
 - and the Window Manager port 497
 - application-defined routines for 671 to 672, 711 to 713
 - checking for
 - when creating a window 500
 - checking for, when zooming windows 686
 - customizations of 666 to 667
 - data structures in 615 to 632, 691 to 694, 724 to 727, 776 to 778
 - direct colors, handling 585 to 587
 - drawing with 591 to 614, 640 to 649
 - indexed colors, handling 583 to 584
 - initializing 589
 - resources in 672 to 676, 713, 816 to 817
 - routines in 633 to 667, 695 to 701, 728 to 752
 - testing for availability 588
 - user interface guidelines for 614
- colors
 - application-defined picking method 810 to 816
 - in color graphics ports 637 to 675
 - determining 649 to 651, 775
 - on grayscale devices 587
 - intermediate 651
- color search functions 671 to 672
- ColorSpec data type 491
- ColorSpec data type 625 to 626
- ColorTable data type
 - . *See also* color tables
- ColorTable data type 626 to 627
- color table resources 674 to 675
- color tables
 - creating 662 to 663, 674 to 675
 - data type for 626 to 627
 - default 663
 - defined 581 to 582
 - disposing of 663
 - modifying 667 to 668
 - resource type for 674 to 675
 - . *See also* color lookup tables
- color window records 483 to 488
- commands, menu. *See* menu commands
- Command-Shift-number key sequences 425
- CommentSpec data type 779
- compacting heap zones 384 to 386
- compaction. *See* heap compaction
- CompactMem function 384 to 385
- CompactMemSys function 385 to 386
- compatibility
 - custom Standard File Package dialog boxes 215
- component connections 34, 93 to 97
- Component data type 69
- ComponentDescription data type 65 to 68, 80 to 82
- component description record 65 to 68, 80 to 82
- component file 60 to 61, 99 to 101, 112
- ComponentFunctionImplemented function 78 to 79
- component identifiers 37, 68 to 69, 70 to 71, 74
- ComponentInstance data type 69
- component instances 34, 68 to 69, 73 to 74
- Component Manager 31 to 115
 - data structures in
 - for applications 65 to 68
 - for components 80 to 83
 - requests to components 46 to 56
 - resources in 107 to 113
 - routines in
 - for applications 69 to 80
 - for components 84 to 104
 - testing for availability 35
- ComponentParameters data type 83
- component parameters record 82 to 83
- component requests 46 to 56

- can do 50
- close 49
- open 47 to 48
- register 51 to 52
- target 52 to 53
- unregister 52
- version 50 to 51
- ComponentResource data type 108 to 113
- component resources 60 to 61, 107 to 113
- components
 - calling 101 to 102
 - capturing 52 to 53, 103 to 104
 - cloning 63
 - closing connections to 40, 75
 - defined 32
 - finding 36 to 37, 70 to 72
 - getting information about 38 to 39, 75 to 79
 - hiding 102 to 104
 - interfaces of, defining 56 to 58
 - levels of service 32, 63, 101
 - manufacturer code for 32, 67, 81
 - opening connections to 35 to 38, 72 to 74
 - registering 58 to 60, 85 to 90, 108
 - requesting services from 46 to 55
 - structure of 41 to 46
 - targeting 53
 - unregistering 52
 - using services of 39 to 40
- ComponentSetTarget function 53, 104 to 105
- component subtypes 32, 66, 81
- component types 32, 66, 81
- concatenating memory blocks 377 to 379
- CopyBits procedure 596 to 598, 718, 721
- CopyDeepMask procedure 600 to 602, 722
- copying memory blocks 306 to 307, 372 to 377
- CopyMask procedure 598 to 600, 722 to 723
- CopyPixmap procedure 656
- CopyPixPat procedure 660
- CountAppFiles procedure 173
- CountComponentInstances function 95
- CountComponents function 71 to 72
- CQDProcs data type 630 to 631
- CreateResFile procedure 165
- creation dates
 - handled by FSpExchangeFiles 140
- cSpecArray data type 625 to 626
- CTabChanged procedure 667 to 668
- CurrentA5 global variable
 - defined 245
 - getting value 311
- CurrentA5 global variable 311
- current device
 - defined 680
 - determining 702
 - setting 700

- current directory
 - in Standard File Package dialog boxes 179, 206 to 209
- current disk. *See* current volume
- current heap zone 318
- current volume 207 to 209
 - in Standard File Package dialog boxes 179, 206 to 209
- cushions. *See* memory cushions
- custom dialog boxes. *See* dialog boxes, custom
- CustomGetFile procedure 226 to 227
- CustomPutFile procedure 221 to 222
- CWindowPeek 28
- CWindowPeek data type 484
- CWindowPtr 28
- CWindowPtr data type 484
- CWindowRecord 28
- CWindowRecord data type 484 to 488

D

- dangling pointers
 - avoiding 261 to 265
 - causes of 261 to 265
 - dangling procedure pointers 264 to 265
 - defined 261
 - detecting 261
 - introduced 252
 - locking blocks to prevent 261 to 262
 - referencing callback routines 264 to 265
 - using local variables to prevent 263
- data buffers 123
- data forks 119, 757
 - creating 165
- 'dctb' resource type 195
- DelegateComponentCall function 63, 64, 101 to 102
- dereferenced handles 261
- desk accessories
 - handling events in 426, 435
- DeskHook global variable
 - clearing in Pascal 321
 - and displaying windows during startup time 321
- DeskHook global variable 498
- DeskPattern global variable 541
- desktop pattern 541
- destination rectangles
 - for the DrawPicture procedure 768 to 769
- DetachResource procedure 325
- DeviceList global variable 680
- device lists
 - defined 680
 - getting first device in 702 to 703
- DeviceLoopFlags data type 694 to 695
- DeviceLoop procedure 684 to 685, 705 to 706
- dialog boxes

- custom 182 to 187, 191 to 206
- displaying file types in 191
- for saving and opening files 179 to 187
 - custom 191 to 206
 - item numbers 197
- resources 192
- standard 179 to 182
- dialog boxes, and low-memory situations 276
- dialog hook functions 196 to 203, 210 to 213, 231 to 232
- Dialog Manager
 - and QuickDraw 576
- direct devices
 - defined 575
 - pixel values for 585 to 587
- directories
 - current 206 to 209
 - defined 124
 - selecting 184 to 187, 209 to 213
- directory IDs
 - defined 124
- disk caches 123
- disk switch dialog box 125
- display list 179
- DisposeCTable procedure. *See* DisposeCTable procedure
- DisposeCTable procedure 663
- disposed handles
 - checking for 265
 - defined 265
 - preventing dereferencing of 265
 - problems using 265
- DisposeGDevice procedure 701
- DisposeGWorld procedure 719, 738
- DisposeHandle procedure 278, 289, 347 to 348
- DisposePictInfo function 809
- DisposePixMap procedure 657
- DisposePixPat procedure 595, 661
- DisposePtr procedure 278, 292, 351
- DisposeScreenBuffer procedure 739
- DisposeWindow procedure 763, 770
- DisposeWindow routine 533 to 534
- DisposPictInfo function. *See* DisposePictInfo function
- DisposPixMap procedure. *See* DisposePixMap procedure
- DisposPixPat procedure. *See* DisposePixPat procedure
- ditherCopy mode 607
- dithering 607 to 608
- ditherPix flag 726, 727, 736, 737
- 'DITL' resource type
 - for default Open and Save dialog boxes 192 to 195
- 'DLOG' resource type
 - for default Open and Save dialog boxes 192 to 193
- document 118

- document records 129
- doesActivateOnFGSwitch flag 472
- dontMatchSeeds flag 706
- double click, time between 459
- double indirection 250
- DragGrayRgn function 521 to 525
- DragHook global variable 525, 527, 530, 532
- DragWindow routine 519 to 520
- DrawGrowIcon routine 511 to 512
- Drawing Environment 29
- DrawNew routine 548
- DrawPicture procedure 762, 768 to 769, 793 to 794
- duplicating relocatable blocks 375 to 377

E

- EmptyHandle procedure
 - used by a grow-zone function 281
- EmptyHandle procedure 299 to 300, 364 to 365
- empty handles
 - checking for 266
 - defined 266
- end-of-file
 - logical 122
 - physical 122
- EndUpdate routine 535
- enhanced Standard File Package routines 188
- EOF. *See* end-of-file
- EraseRect procedure 605, 686, 723
- EraseRect routine 530
- error handling
 - for Color QuickDraw routines 664 to 665
- EventAvail function 427 to 428
- Event Manager
 - application-defined routine for 467 to 468
 - data structures in 407 to 420
 - routines in 421 to 461
- event masks
 - defined 409
- event messages 415
- event queue
 - . *See also* high-level event queue
 - . *See also* Operating System event queue
 - structure of 419 to 420
- EventRecord 28
- EventRecord data type 414 to 417
- event records 414 to 417
- events
 - kinds of 407
- EvQEl data type 420
- ext32Device flag 693, 699, 707, 712
- extended version 2 format 755 to 756, 786 to 788

F

-
- fake handles 267 to 268, 287, 342
 - creating 267, 268
 - defined 267
 - problems using 267, 287, 342
 - file control blocks 123
 - file data 120
 - limitations of using Resource Manager 120
 - using the File Manager to read 120
 - using the Resource Manager to read 120
 - file filter functions
 - for file display list 195, 230 to 231
 - file forks
 - data fork 119
 - resource fork 119
 - file formats in Standard File Package dialog boxes 183 to 184
 - file fragmentation 122
 - File Manager
 - creating FSSpec records 168 to 169
 - exchanging contents of two files 167
 - testing for features 128
 - file marks 123
 - File menu
 - adjusting items in 151 to 152
 - appearance of 126
 - Close command 146 to 148
 - New command 130
 - Open command 132 to 136, 188
 - Revert to Saved command 144 to 146
 - Save As command 140 to 144, 188
 - Save command 140 to 144, 188
 - user selections in 127, 130 to 148
 - file permissions 135
 - file reference numbers
 - defined 123
 - files
 - adjusting size of 122, 162
 - closing 146 to 148
 - creating 130 to 132, 165
 - defined 118
 - exchanging data in 167
 - handling File menu commands 127
 - opening 132 to 136
 - Standard File Package 156 to 157, 179 to 180, 184, 224 to 229
 - opening at application startup 148 to 150
 - permissions 135
 - reading data 136 to 137, 159
 - reading data in newline mode 123
 - reverting to last saved version 144 to 146
 - saving 140 to 144, 180 to 181, 182 to 183
 - saving preferences 150 to 151
 - saving under a new name 141
 - user interface for saving and opening 177
 - writing data 137 to 140
 - File System Specification Record 26
 - file system specification records
 - creating 165 to 166
 - with Standard File Package 189
 - file types, filtering Standard File Package display lists
 - by 225
 - FillCArc procedure 646
 - FillCOval procedure 645
 - FillCPoly procedure 646 to 647
 - FillCRect procedure 595, 644
 - FillCRgn procedure 647
 - FillCRoundRect procedure 644 to 645
 - fill patterns
 - in color graphics ports 644 to 647
 - FillRect procedure 592
 - fills
 - calculating color 652 to 654
 - filter function
 - for GetSpecificHighLevelEvent 431 to 432, 467 to 468
 - filter functions
 - modal-dialog filter functions 203 to 205
 - with Standard File Package routines 191, 195 to 196
 - Finder, allocation of memory for disk copying 322
 - FindFolder function 128
 - FindNextComponent function 36, 70 to 71
 - FindWindow function 517 to 518
 - 'FKEY' resource type 425
 - FlushEvents procedure 433 to 434
 - flushing a volume 138, 148, 169 to 170
 - FlushVol function 148, 169 to 170
 - folders. *See* directories
 - FontSpec data type 779 to 781
 - foreground colors 591 to 593, 640 to 641, 649
 - formats for pictures
 - extended version 2 755 to 756, 786 to 788
 - version 1 755 to 756
 - version 2 755 to 756, 788
 - fragmentation. *See* heap fragmentation
 - FreeMem function 379 to 380
 - FreeMemSys function 380
 - free space
 - assessing 379 to 383
 - assessing availability for temporary memory 392 to 393
 - FrontWindow function 518
 - FSClose function 159 to 160
 - FSMakeFSSpec function 168 to 169
 - FSpCreate function 165 to 166
 - FSpCreateResFile procedure 165
 - FSpDelete function 166
 - FSpExchangeFiles function 139 to 140, 167
 - FSpOpenDF function 164, 764

FSOpenResFile function 165
 FSRead function 158
 FSSpec 26
 FSSpec data type 126, 153
 FSWrite function 159

G

gaps in heaps, danger of 257
 gdDevType flag 693, 699, 707, 709, 710, 712
 GDevice 29
 GDeviceChanged procedure 670
 GDevice data type
 . *See also* graphics devices
 GDevice data type 691 to 694
 GDevice records
 creating 696 to 699
 disposing of 701
 getting available 701 to 704
 modifying 670
 . *See also* graphics devices
 setting attributes for 698 to 699
 setting for current device 700
 with greatest pixel depth 703 to 704
 gestaltQuickDrawFeatures selector 589
 gestaltQuickDrawVersion selector 588
 GetAppFiles procedure 173
 GetApplLimit function 285, 397
 GetAppParms procedure 172
 GetAuxWin function 545 to 546
 GetBackColor procedure 650
 GetCaretTime function 460 to 461
 GetComponentIconSuite function 77 to 78
 GetComponentInfo function 38 to 39, 76 to 77
 GetComponentInstanceA5 function 96 to 97
 GetComponentInstanceError function 79 to 80
 GetComponentInstanceStorage function 95
 GetComponentListModSeed function 72
 GetComponentRefcon function 63, 99
 GetComponentVersion function 78
 GetCPixel procedure 650 to 651
 GetCTable function 662 to 663
 GetCWMgrPort routine 543
 GetDblTime function 458 to 459
 GetDeviceList function 687, 702 to 703
 GetEOF function 162
 GetEvQHdr function 442
 GetForeColor procedure 649
 GetFPos function 160 to 161
 getFrontClicks flag 472
 GetGDevice function 702
 GetGray function 651
 GetGrayRgn function 542

GetGWorldDevice function 741 to 742
 GetGWorldPixmap function 718, 743 to 744
 GetGWorld procedure 718, 739 to 740
 GetHandleSize function 352 to 353
 GetKeys procedure 455 to 456
 GetMainDevice function 687, 703
 GetMaxDevice function 703 to 704
 GetMouse procedure 453
 GetNewCWindow function 498 to 500
 GetNewCWindow function 590
 GetNewWindow function 501 to 502
 GetNewWindow function 590
 GetNextDevice function 687, 704
 GetNextEvent function 428 to 429
 GetNextEvent function, and temporary memory 322
 GetOSEvent function 438 to 439
 GetPictInfo function 774, 796 to 799
 GetPicture function 795
 GetPixBaseAddr function 750 to 751
 GetPixelsState function 748 to 749
 GetPixmapInfo function 799 to 801
 GetPixPat function 595, 658
 GetPortNameFromProcessSerialNumber
 function 451 to 452
 GetProcessSerialNumberFromPortName
 function 418, 450 to 451
 GetPtrSize function 354 to 355
 GetSpecificHighLevelEvent function 431 to 433
 GetVInfo function 170
 GetVRefNum function 171
 GetWindowPic function 538 to 539
 GetWindowPic function 763
 GetWMgrPort routine 543 to 544
 GetWRefCon function 539
 GetWTitle routine 510 to 511
 GetWVariant function 540
 GetZone function 393
 global variables
 DeviceList 680
 HiliteRGB 612
 MainDevice 703
 QDColors 641
 ScrHRes 708
 ScrVRes 708
 TheGDevice 680
 global variables. *See* application global variables;
 system global variables; QuickDraw global
 variables
 GrafPort records
 and color pictures 756 to 757
 compared with CGrafPort records 578 to 579
 copying images from offscreen graphics worlds 721
 to 723
 getting 720, 739 to 740
 restoring 720, 740 to 741

- saving 720, 739 to 740
- setting 720, 740 to 741
- GrafVars data type 632
- graphics device records. *See* GDevice records
- Graphics Devices 29
- graphics devices 679 to 713
 - application-defined routine for 711 to 713
 - data structures in 691 to 694
 - defined 679
 - determining characteristics of 684 to 685, 705 to 708
 - getting handles to 701 to 704
 - optimizing images for 684 to 689, 705 to 706, 711 to 713
 - resource for 713
 - routines for 695 to 701
 - . *See also* GDevice records
 - testing for availability 684
 - with greatest pixel depth 703 to 704
- graphics pens
 - attributes of 622
 - colors for 591 to 596, 637 to 638, 640 to 641
 - pixel patterns for 637 to 638
- graphics ports
 - data types for 618 to 624
 - modifying 669 to 670
 - restoring 720, 739 to 741
 - saving 720, 739 to 741
 - setting 720, 739 to 741
- GrayRgn global variable 519, 542, 547
- grayscale devices
 - colors on 587
- grow images, of windows 512
- GrowWindow function 526 to 527
- grow-zone functions 280 to 281, 312 to 313, 402 to 403
 - defined 270
 - example of 281
 - finding protected block 310, 390
 - setting 309 to 310, 389 to 390
 - using SetA5 function 313, 403
 - using SetCurrentA5 function 313, 403
- gwFlagErr flag 726
- GWorldFlags 30
- GWorldFlags data type 725 to 727
- GWorldPtr data type 724
- GWorld. *See* offscreen graphics worlds
- GZRootHnd global variable 310, 390
- GZSaveHnd function 281, 310, 390

H

- HandAndHand function 377 to 378
- Handle data type 250, 337
- handles

- .*See also* relocatable blocks
- checking validity of 266
- defined 250 to 251
- recovering 367 to 368
- relative 335
- HandleZone function 395 to 396
- HandToHand function 375 to 377
- HasDepth function 689, 709 to 710
- HClrRBit procedure 363 to 364
- HCreateResFile procedure 165
- 'hdlg' resource type 194
- heap compaction
 - defined 243, 255
 - movement of relocatable blocks during 256
 - routines for 384 to 386, 387 to 389
- heap fragmentation
 - causes of 257 to 260
 - defined 242
 - during memory reservation 257
 - maximizing heap size to prevent 272
 - preventing 256 to 260
 - summary of prevention 260
- heap purging 253 to 254
 - routines for 386 to 389
- heap. *See* application heap; system heap
- heap zones
 - accessing 393 to 396
 - changing 394
 - defined 317
 - getting current zone 393
 - initializing 399 to 400
 - manipulating 396 to 402
 - organization of 332 to 334
 - .*See also* zone headers; zone trailers
 - subdividing into multiple heap zones 327 to 329
- HFS volumes
 - defined 124
- HGetState function 262, 293 to 294, 356 to 357
- HideWindow routine 513 to 514
- hierarchical file system (HFS)
 - defined 123 to 125
- high-level event message record 418 to 419
- HighLevelEventMsg data type 419
- high-level events
 - receiving 425
 - searching for a specific event 431
 - sending 443
- highlighting 611 to 614, 648 to 649
- high memory, allocating at startup time 326 to 327
- HiliteColor procedure 648 to 649
- hilite mode 614
- HiliteRGB global variable 612
- HiliteWindow routine 515
- histograms 810, 812 to 813
- HLockHi procedure 305, 371 to 372

HLock procedure 262, 295 to 296, 358 to 359
 HNoPurge procedure 298 to 299, 361 to 362
 HOpenResFile function 165
 HPurge procedure 297 to 298, 360 to 361
 HSetRBit procedure 362 to 363
 HSetState procedure 262, 294 to 295, 357 to 358
 HUnlock procedure 296 to 297, 359 to 360

I

icons
 for components 76, 87, 109
 images
 copying 596 to 602, 721 to 723
 indexed devices
 defined 575
 pixel values for 583 to 584
 InitApplZone procedure 400 to 401
 InitCPort procedure 636
 InitGDevice procedure 697 to 698
 initializing new heap zones within other heap
 zones 327 to 329
 InitWindows routine 497 to 498
 InitZone procedure 313, 399 to 400, 403
 interprocess buffers, and temporary memory 322
 interrupt tasks
 and Memory Manager routines 282, 339
 and temporary memory 322
 interrupt time
 avoiding Memory Manager routines at 282, 339
 InvalRect routine 535 to 536
 InvalRgn routine 536
 inverse tables
 defined 681
 is32BitCompatible flag 473
 isHighLevelEventAware flag 473
 isStationeryAware flag 474

J

jump table 245
 jump table entries
 for callback routines 264

K

keepLocal flag 725, 726, 730, 732, 736
 keyboard equivalents, in Standard File Package dialog
 boxes 182

keyboards
 getting the state of 455
 KeyMap data type 455
 KeyTrans function. *See* KeyTranslate function
 KeyTranslate function 456 to 457
 KillPicture procedure 763, 791 to 792

L

linked lists, allocating new elements in 263
 loading code segments, and dangling pointers 263 to 264
 localAndRemoteHLEvents flag 473
 locking relocatable blocks 252 to 253, 295 to 296, 358 to 359
 LockPixels function 718, 744 to 745
 logical blocks 120
 logical end-of-file 121 to 122
 logical sizes of blocks 334
 low-memory conditions 268 to 270
 low-memory global variables
 See system global variables
 luminance 587

M

Mac OS Events 28
 Mac OS File System 26
 MainDevice global variable 703
 main screen
 determining 703
 mainScreen flag 693, 699, 707, 712
 MakeRGBPat procedure 660 to 661
 manufacturer code for components 32, 67, 81
 mapPix flag 725, 727, 737
 marks. *See* file marks
 master pointer blocks 250
 master pointers
 allocating manually 283 to 284, 340 to 341
 defined 250
 determining how many to preallocate 273 to 274
 number per block in application zone 273
 running out of 273
 MatchRec data type 627
 MaxApplZone procedure
 and ApplLimit global variable 240
 automatic execution of 272, 329
 and heap fragmentation 272
 MaxApplZone procedure 283, 339 to 340
 MaxBlock function 380 to 381
 MaxBlockSys function 381

- maximizing heap zone space 387 to 389
- MaxMem function 387 to 388
- MaxMemSys function 388 to 389
- maxSize constant 385
- MC680x0 microprocessor
 - size of memory blocks with 334
- MemErr global variable 282, 308, 339, 384
- MemError function 282, 308, 339, 383 to 384
- memory
 - allocating and releasing 286 to 292, 341 to 351
 - allocating during startup 326 to 327
 - assessing 379 to 396
 - changing sizes of blocks 351 to 356
 - freeing 384 to 389
 - organization of 236 to 245, 331 to 336
 - .See also temporary memory; virtual memory
- memory blocks. See blocks, memory
- memory cushions
 - defined 269
 - determining optimal size of 275
 - maintaining 275 to 276
- Memory Manager 315 to 405
 - 24-bit 247
 - 32-bit 247
 - allocating master pointers 273
 - and application heap 242 to 243
 - application-defined routines 402 to 405
 - calling grow-zone function 280
 - capabilities of 316
 - compacting heap 255 to 256
 - data types 249 to 250, 337 to 338
 - defined 316
 - movement of blocks by 256
 - purging heap 255 to 256
 - reserving memory 254 to 255, 368 to 369
 - returning result codes 282, 308, 339, 383 to 384
 - routines 338 to 402
 - testing for features 323 to 324
- memory reservation. See reserving memory
- memory reserves
 - benefits of 269
 - defined 269
 - maintaining 278 to 280
- MemTop global variable 326, 399
- menu commands
 - Close (File menu) 126 to 128, 146 to 148
 - New (File menu) 126 to 128, 130 to 132
 - Open (File menu) 126 to 128, 132 to 136
 - Revert to Saved (File menu) 126 to 128, 144 to 146
 - Save (File menu) 126 to 128, 140 to 144
 - Save As (File menu) 126 to 128, 140 to 144
- modal-dialog filter functions, for Standard File Package
 - dialog boxes 198, 203 to 205, 232 to 234
- modification dates, handled by
 - FSpExchangeFiles 140

- modifier keys 416
- MoreMasters procedure 273 to 274, 283 to 284, 340 to 341
- mouse
 - determining location of 453
 - getting information about 453
- mouse-down events
 - in desk accessories 426, 435
- mouse-moved events 424
- MoveHHi procedure 258 to 259, 303 to 304, 369 to 371
- MoveWindow routine 520
- moving relocatable blocks high 258 to 259, 303 to 305, 369 to 372
- multiple heap zones
 - implementing 327 to 329
 - uses for 318 to 319
- MyCalcColorTable function 814 to 815
- MyColorSearch function 671 to 672
- MyDisposeColorPickMethod function 816
- MyDrawingProc procedure 712 to 713
- MyInitPickMethod function 811 to 813
- MyRecordColors function 813 to 814

N

- New command (File menu) 126 to 128, 130 to 132
- NewCWindow function 503 to 506
- newDepth flag 725, 727, 737
- NewEmptyHandle function 345 to 346
- NewEmptyHandleSys function 346 to 347
- New Folder dialog box 143, 154, 181, 217
- NewGDevice function 696 to 697
- NewGWorld function 717 to 719, 728 to 732
- NewHandleClear function 277, 288, 344
- NewHandle function 276, 287 to 288, 342 to 343
- NewHandleSysClear function 344 to 345
- NewHandleSys function 343 to 344
- newline character 123
- newline mode 123
- NewPictInfo function 802 to 804
- NewPixMap function 655 to 656
- NewPixPat function 658 to 659
- NewPtrClear function 291, 350
- NewPtr function 276, 290 to 291, 348 to 349
- NewPtrSysClear function 350 to 351
- NewPtrSys function 349 to 350
- newRowBytes flag 725, 727, 737
- NewScreenBuffer function 733
- NewTempScreenBuffer function 734
- NewWindow function 506 to 509
- noDriver flag 693, 699, 707, 712
- nonessential memory requests, checking whether to satisfy 275

noNewDevice flag 725, 726, 730, 732, 742
 nonrelocatable blocks
 .See also blocks, memory
 advantages of 252
 allocating 260, 290 to 291, 348 to 351
 allocating temporarily 260
 data type for 250
 defined 249
 disposal and reallocation of 257
 releasing 292, 351
 sizing 354 to 356
 when to allocate 259 to 260
 NoPurgePixels procedure 747
 notSrcCopy source mode 603, 604
 notSrcOr source mode 603, 604
 notSrcXor source mode 603

O

offline volumes 125
 offscreen graphics worlds 715 to 752
 copying images from 721 to 723
 creating 717 to 719, 728 to 734
 data structures in 724 to 727
 defined 715
 disposing of 738 to 739
 drawing into 720 to 721
 restoring 720, 739 to 741
 routines for 728 to 752
 saving 720, 739 to 741
 setting 720, 739 to 741
 testing for availability 717
 updating 721, 735 to 738
 OldContent global variable 548
 OldStructure global variable 548
 onlyBackground flag 472
 opcodes 756
 OpColor procedure 648
 Open command (File menu) 126 to 128, 132 to 136
 OpenComponent function 74
 OpenComponentResFile function 99 to 100
 OpenCPicParams records 778
 OpenCPicture function 761, 786 to 788
 OpenCPort procedure 634 to 635
 OpenDefaultComponent function 35 to 36, 73 to 74
 opening files
 at application startup 148 to 150
 with Standard File Package 179 to 180, 184
 OpenPicture function 788 to 789
 open request 47 to 48
 OpenResFile function 165
 Operating System event queue
 flushing events from 433

operating system queues, storing elements in system
 heap zone 324 to 325
 original application heap zone 317
 original Standard File Package procedures 215 to 216,
 218
 OSEventAvail function 439 to 441
 ovals
 filling
 with pixel patterns 645

P

PaintBehind routine 549 to 550
 PaintOne routine 548 to 549
 PaintRect procedure 592, 595
 PaintWhite global variable 549, 550
 parent directories 125
 parent directory IDs 125
 partitions 236
 .See also application partitions; system partition
 sizes of 469
 path reference numbers. *See* file reference numbers
 pattern modes 603
 patterns
 background, in color graphics ports 638 to 639
 changing 638 to 639
 data types for 628 to 630
 fill, in color graphics ports 644 to 647
 of graphics pens in color graphics ports 637 to 638
 resources for 673
 stretching for printer output 623
 pCDeskPat parameter-RAM bit flag 541
 PenPixPat procedure 637 to 638
 physical end-of-file 121 to 122
 physical sizes of blocks 334
 PicComment procedure 789 to 791
 'PICT' file type 757, 763 to 766, 770 to 773
 PictInfo data type 781 to 785
 'PICT' resource type 757 to 758, 769 to 770, 795, 816 to
 817
 'PICT' scrap format 758, 767, 771
 picture comments 789 to 791
 defined 756
 inserting into pictures or printing code 789 to 791
 Picture data type
 . See also pictures
 Picture data type 776 to 777
 Picture Records 30
 picture resources 757 to 758, 769 to 770, 795, 816 to 817
 Pictures 30
 pictures
 collecting information from 773 to 775, 795 to 799,
 802 to 806, 807 to 809

- color, in basic graphics ports 756 to 757
- creating 760 to 763, 786 to 791
- data type for 776 to 777
- defined 754
- destination rectangles for 768 to 769
- disposing of 763, 770, 791 to 792
- drawing 760 to 770, 792 to 794
- extended version 2 format 755 to 756, 786 to 788
- opcodes for 756
- opening 763 to 770
- in 'PICT' files 757, 763 to 766, 770 to 773
- in 'PICT' resources 757 to 758, 769 to 770, 771, 795
- reading from a resource file 795
- resolutions for 761, 769
- saving 770 to 773
- in the scrap 758, 767, 771
- version 1 format 755 to 756
- version 2 format 755 to 756, 788
- and the Window Manager 763
- Picture Utilities
 - application-defined routines for 810 to 816
 - data structures in 779 to 785
 - defined 758
 - gathering information with 773 to 775
 - routines in 795 to 809
 - testing for availability 760
- PinRect function 525 to 526
- pixel depths
 - default color tables for 663
 - defined 580
 - determining 684 to 689, 705 to 706, 709 to 710
 - setting 689, 710 to 711
- pixel images
 - addresses of, for offscreen graphics worlds 750 to 751
 - defined 580 to 582
 - getting states of, for offscreen graphics worlds 748 to 749
 - locking, for offscreen graphics worlds 744 to 745
 - in pixel maps 580 to 582
 - purgeable, for offscreen graphics worlds 746 to 747
 - setting states, for offscreen graphics worlds 749 to 750
 - unlocking, for offscreen graphics worlds 745 to 746
 - unpurgeable, for offscreen graphics worlds 747
 - whether in 32-bit mode, for offscreen graphics worlds 751 to 752
- pixel maps
 - copying images between 596 to 602
 - creating 655 to 656
 - data type for 616 to 618
 - defined 579
 - disposing of 657
 - gathering color information from 799 to 804, 806 to 809
 - obtaining, for offscreen graphics worlds 743 to 744
 - pixel images in 580 to 582
 - setting 656 to 657
- pixel pattern resources 594 to 595, 673
- pixel patterns
 - background 638 to 639
 - creating 658 to 661, 673
 - data type for 628 to 630
 - defined 582 to 583
 - disposing of 661
 - filling with 593 to 596, 644 to 647
 - framing and painting with 593 to 596
 - of graphics pens 593 to 596, 637 to 638
 - modifying 668 to 669
 - resources for 594 to 595, 673
- pixels
 - colors for
 - in Color QuickDraw 574 to 575, 580 to 581, 583 to 587, 591 to 614
 - copying between pixel maps 596 to 602
 - copying from offscreen graphics worlds 721 to 723
 - depths of. *See* pixel depths
 - patterns for. *See* bit patterns, pixel patterns
 - values for. *See* pixel values
- pixelsLocked flag 725, 727, 748, 749
- pixelsPurgeable flag 725, 726, 748, 749
- pixel values
 - as RGB colors 583 to 587
 - defined 581
 - for direct devices 585 to 587
 - for indexed devices 583 to 584
- PixelFormat32Bit function 751 to 752
- PixelFormat data type
 - . *See also* pixel maps
- PixelFormat data type 616 to 618
- PixelFormat records
 - creating 655 to 656
 - disposing of 657
 - obtaining, for offscreen graphics worlds 743 to 744
 - pixel images in 580 to 582
 - setting 656 to 657
- PixPatChanged procedure 668 to 669
- PixPat data type
 - . *See also* pixel patterns
- PixPat data type 628 to 630
- PixPatHandle data type 628
- pixPurge flag 725, 726, 730, 731
- pointers 249 to 250
 - . *See also* nonrelocatable blocks; dangling pointers
- polygons
 - filling
 - with pixel patterns 646 to 647
- pop-up menus
 - in Standard File Package dialog boxes 183
- PortChanged procedure 669 to 670

- port names
 - converting to process serial numbers 450
- port rectangle 528, 530
 - of Window Manager port 497
- port rectangles
 - in color graphics ports 621
- PostEvent function 448 to 449
- PostHighLevelEvent function 443 to 446
- 'ppat' resource type 594 to 595, 673
- PPostEvent function 446 to 448
- preferences files 118, 150 to 151
- Preferences folder 125
- process serial numbers, converting to port names 450
 - to 452
- ProcPtr data type
 - and code segmentation 264 to 265
 - referencing code in code resources 325
- ProcPtr data type 337 to 338
- protected blocks
 - defined 281
 - determining which they are 313, 403
 - handle to returned by GZSaveHnd 310, 390
- pseudo-items 197 to 201
 - constant descriptions 197 to 202
- PtrAndHand function 378 to 379
- Ptr data type 249, 337
- PtrToHand function 373 to 374
- PtrToXHand function 374 to 375
- PtrZone function 396
- PurgeMem procedure 386 to 387
- PurgeMemSys procedure 387
- PurgeSpace procedure 307, 381 to 382
- purge-warning procedures 329 to 331, 334, 403 to 405
 - defined 329
 - installed by SetResPurge 331, 404
 - restrictions on 404
 - sample 329
 - using SetA5 function 404
 - using SetCurrentA5 function 404
- purging heap zones 256, 386 to 387
- purging relocatable blocks 253 to 254

Q

- QDCOLOR global variable 641
- QDError function 664 to 665, 770
- QHDr data type 419
- QuickDraw 28
 - customizations of 666 to 667
 - and Dialog Manager 576
- QuickDraw global variables
 - defined 245
- QuickDraw Operations 29

R

- radio buttons, in Standard File Package dialog
 - boxes 183
- ramInit flag 693, 699, 707, 712
- reading data from files 136 to 137, 158
- ReallocateHandle procedure 300 to 301, 365 to 366
- reallocating relocatable blocks 253 to 254
- reallocPix flag 726, 727, 737
- RecordPictInfo function 805 to 806
- RecordPixMapInfo function 806 to 807
- RecoverHandle function 367 to 368
- rectangles
 - filling
 - with pixel patterns 644
- refCon field 539
- regions
 - filling
 - with pixel patterns 647
- RegisterComponent function 58, 85 to 87
- RegisterComponentResourceFile function 89 to 90
- RegisterComponentResource function 58, 87 to 89
- register request 51 to 52
- relative handles 335
- relocatable blocks
 - .See also blocks, memory; handles
 - allocating 287 to 288, 342 to 347
 - changing properties 292 to 299, 356 to 364
 - clearing resource bit 363 to 364
 - concatenating 377 to 378
 - data type for 249
 - defined 249
 - disadvantages of 252
 - duplicating 375 to 377
 - emptying 299 to 300, 364 to 365
 - getting properties 293 to 294, 356 to 357
 - in bottom of heap zone 257
 - locking 252 to 253, 295 to 296, 358 to 359
 - for long periods of time 260
 - for short periods of time 260
 - making purgeable 297 to 298, 360 to 361
 - making un purgeable 298 to 299, 361 to 362
 - managing 299 to 305, 364 to 372
 - master pointers after disposing 265
 - master pointers for 273
 - moving around nonrelocatable blocks 256
 - moving high 258 to 259, 303 to 305, 369 to 372
 - properties of 252 to 254
 - purging 253 to 254
 - reallocating 253 to 254, 300 to 301, 365 to 366
 - releasing 289, 347 to 348
 - restrictions on locked blocks 259
 - setting properties 294 to 299, 357 to 364
 - setting resource bit 362 to 363
 - sizing 352 to 354

- movement during 256
 - unlocking 252 to 253, 296 to 297, 359 to 360
 - when to lock 260
- reply records for Standard File Package 188 to 189, 216 to 219
- request codes
 - for components 47, 57
- ReserveMem procedure 302 to 303, 368 to 369
- ReserveMemSys procedure 369
- reserves. *See* memory reserves
- reserving memory 254 to 255
 - and heap fragmentation 257
 - defined 254
 - for relocatable blocks 258
 - limitation of 257
 - routines 368 to 369
- resolutions
 - for screens 708
 - for pictures 761, 769
- resource bit
 - clearing 363 to 364
 - setting 362 to 363
- resource editors 193
- resource forks 119, 757
 - creating 165
 - creating resource map in 165
- Resource Manager, installing purge-warning procedures 331, 404
- resources
 - color icon 675 to 676
 - color-picking method 817
 - color table 674 to 675
 - component 61, 89 to 90, 107 to 113
 - picture 757 to 758, 769 to 770, 795, 816 to 817
 - pixel pattern 594 to 595, 673
 - screen 713
 - size 468 to 474
 - window 568 to 571
 - window color table 571
 - window definition function 563, 571
- ResourceSpec data type 109
- resource types
 - 'cicn' 675 to 676
 - 'clut' 674 to 675
 - 'cmpt' 817
 - 'dctb' 195
 - 'DITL' 143, 181
 - 'DLOG' 192
 - 'hdlg' 194
 - 'PICT' 757 to 758, 769 to 770, 795, 816 to 817
 - 'ppat' 594 to 595, 673
 - 'scrn' 713
 - 'SIZE' 245, 468 to 474
 - 'sysz' 326
 - 'thng'. *See* 'thng' resource type

- 'wctb' 571
 - 'WDEF' 563, 571
 - 'WIND' 568 to 571
- result codes for Memory Manager routines 282, 308, 339, 383 to 384
- RetrievePictInfo function 807 to 808
- Revert to Saved command (File menu) 126 to 128, 144 to 146
- Rez 192
- RGBBackColor procedure 642 to 643
- RGBColorArray data type 813
- RGBColor data type
 - . *See also* RGB colors
- RGBColor data type 625
- RGBColor records 583 to 587
- RGB colors
 - as pixel values 583 to 587
 - data type for 625
 - defined 574 to 575
- RGBForeColor procedure 592, 640 to 641
- root directory 124
- rounded rectangles
 - filling
 - with pixel patterns 644 to 645

S

- sample routines
 - DoNew 590
 - DoSavePictAsCmd 770
 - DoUpdate 684
 - DoZoomWindow 686 to 688
 - DrawerSetup 56
 - HiliteDemonstration 613
 - MyAdjustDestRect 768
 - MyCopyBlackAndRedMasks 722
 - MyCreateAndDrawPict 761
 - MyDrawFilePicture 763
 - MyDrawResPict 769
 - MyFileGetPic 766
 - MyFilePutPic 773
 - MyFindVideoComponent 37
 - MyGetCompInfo 39
 - MyGetComponent 38
 - MyGetPictProfileCount 774
 - MyIsColorPort 766
 - MyPaintAndFillColorRects 592
 - MyPaintPixelPatternRects 595
 - MyPaintRectsThruGWorld 718
 - MyPastePict 767
 - MyReplaceGetPic 765
 - MyReplacePutPic 772
 - MySetHiliteMode 612

- MyTrivialDrawingProc 685
- OvalCanDo 50
- OvalClick 55
- OvalClose 49
- OvalDraw 54
- OvalDrawer 44 to 46
- OvalErase 55
- OvalMoveTo 56
- OvalOpen 48
- OvalSetUp 54
- RectangleDrawer 64
- Save As command (File menu) 126 to 128, 140 to 144
- Save command (File menu) 126 to 128, 140 to 144
- SaveOld routine 547 to 548
- SaveUpdate global variable 549
- saving files 140 to 144
- saving to different file formats 183
- scrap
 - defined 758
 - pictures in 758, 767, 771
- screenActive flag 693, 699, 707, 712
- screenDevice flag 693, 699, 707, 712
- screen resources 713
- ScreenRes procedure 708
- screens
 - determining characteristics of 705 to 708
 - optimizing images for 684 to 689, 705 to 706, 711 to 713
 - resolution of 708
 - with greatest pixel depth 703 to 704
- ScrHRes global variable 708
- scripts
 - specifying when creating a file 165
- 'scrn' resource type 713
- ScrVRes global variable 708
- SectRect function 687
- SeedCFill procedure 652 to 653
- SelectWindow routine 512
- SendBehind routine 516 to 517
- SetA5 function
 - used in a grow-zone function 313, 403
 - used in a purge-warning procedure 404
- SetA5 function 311
- SetApplBase procedure 401 to 402
- SetApplLimit procedure
 - using to increase size of stack 272
- SetApplLimit procedure 285 to 286, 397 to 398
- SetComponentInstanceA5 procedure 96
- SetComponentInstanceError procedure 56, 97 to 98
- SetComponentInstanceStorage procedure 47, 94 to 95
- SetComponentRefcon procedure 63, 98
- SetCPixel procedure 643
- SetCurrentA5 function
 - used in a grow-zone function 313, 403
 - used in a purge-warning procedure 404
- SetCurrentA5 function 311
- SetDefaultComponent function 105 to 106
- SetDepth function 689, 710 to 711
- SetDeskCPat routine 540 to 541
- SetDeviceAttribute procedure 698 to 699
- SetEOF function 122, 162 to 163
- SetEventMask procedure 441 to 442
- SetFPos function 161
- SetGDevice procedure 700
- SetGrowZone procedure 309 to 310, 313, 389 to 390, 403
- SetGWorld procedure 718, 740 to 741
- SetHandleSize procedure 353 to 354
- SetPixelsState procedure 749 to 750
- SetPortPix procedure 656 to 657
- SetPtrSize procedure 355 to 356
- SetRect procedure 687
- SetResPurge procedure, installing purge-warning procedures 331
- SetStdCProcs procedure 666 to 667, 765, 772
- SetWinColor routine 544 to 545
- SetWindowPic procedure 763, 770
- SetWindowPic routine 538
- SetWRefCon routine 539
- SetWTitle routine 510
- SetZone procedure 394
- SFGetFile procedure 228
- SFGetFile procedure 229
- SFPPutFile procedure 223 to 224
- SFPutFile procedure 222 to 223
- SFReply data type 218
- short-circuit Boolean operators 266
- ShowHide routine 514 to 515
- ShowWindow routine 513
- SignedByte data type 249, 337
- singleDevices flag 706
- size correction for blocks 335, 336
- Size data type 338
- size resources 469 to 474
- 'SIZE' resource type
 - defined 468 to 474
 - flags, defined 470 to 474
- 'SIZE' resource type 468 to 474
- 'SIZE' resource type, specifying partition size 245
- SizeWindow routine 528
- source modes 602 to 607
- srcBic source mode 603, 604, 611
- srcCopy source mode 603, 610
- srcOr source mode 603 to 604, 611
- srcXor source mode 603, 611
- stack
 - collisions with the heap 240
 - default size of 272
 - defined 240
 - determining available space 382

- increasing size of 271 to 272
- stack frame 241
- stack sniffer 240
- StackSpace function 382 to 383
- Standard File Package 177
 - activation procedures 205 to 206, 234
 - and aliases 189
 - application-defined routines in 230 to 234
 - callback routines 195 to 206
 - compatibility with earlier procedures 215 to 216
 - data structures in 216 to 219
 - dialog hook functions 196 to 203, 231 to 232
 - file filter functions 195 to 196, 230 to 231
 - modal-dialog filter functions 203 to 205, 232 to 234
 - opening files 156 to 157, 179 to 180, 224 to 229
 - original procedures 215 to 216
 - original reply record 218 to 219
 - reply records 153 to 155, 188 to 189, 217 to 219
 - routines in 219 to 229
 - saving files 157, 180 to 189, 219 to 224
 - testing for features 188
 - user interface guidelines 187
 - user interfaces
 - custom 182 to 187
 - standard 179 to 182
- StandardFileReply data type 189, 217
- Standard File Reply Records 27
- StandardGetFile procedure 156 to 157, 179, 189, 225, 764
- StandardPutFile procedure 157, 180, 220
- standard state of a window 490, 686
- startup process
 - allocating memory during 326 to 327
 - displaying windows during 321
- stationery pads
 - handled by Standard File Package 154
 - recognition of 474
- stationery pads, handled by Standard File Package 218
- StdPutPic procedure 764
- StillDown function 454
- Str255 data type 337
- stretchPix flag 726, 727, 736, 737
- StringHandle data type 337
- StringPtr data type 337
- subdirectories 124
- subOver arithmetic transfer mode 609, 610
- subPin arithmetic transfer mode 608, 610, 648
- SysEqu.p interface file 319
- SystemClick procedure 426, 435 to 436
- SystemEvent function 437
- system event masks 441
- system extensions, allocating memory at startup
 - time 326
- System Folder 125
- system global variables

- changing 321
- defined 238 to 239, 319
- reading 321
- uses of 319
- system heap 238
 - defined 238
- system heap zone
 - allocating memory in 324 to 325
 - creating new heap zones within 328
 - defined 317
 - getting a pointer to 395
 - installing interrupt code into 325
 - uses for 318
- system partition 236 to 239
 - .See also* system heap; system global variables
- system software version 7.0 188, 201, 204, 215
- SystemTask procedure 425, 436
- SystemZone function 395
- SysZone global variable 395
- 'sysz' resource type 326

T

- tag bytes 335
- TargetID data type 417
- target ID records 417 to 418
- target request 52 to 53
- TempFreeMem function 392
- TempMaxMem function 392 to 393
- TempNewHandle function 391
- temporary memory
 - allocating 323
 - confirming success of allocation 323
 - defined 245, 317
 - determining zone of 322
 - limitation on locking 322
 - operating on blocks 317
 - optimal usage of 317
 - release of during application termination 322
 - routines 390 to 393
 - testing for features of 323 to 324
 - tracking of 322
 - using as a heap zone 329
- TestDeviceAttribute function 687, 707 to 708
- text
 - in color graphics ports 623
- TheGDevice global variable 680
- TheZone global variable 393
- 32-bit addressing
 - defined 247
- 32-bit clean 248
- 'thng' resource type
 - format of 107 to 113

- Rez input for 61
- THz data type 332, 333
- TickCount function 458
- Ticks global variable 458
- TopMem function 327, 398 to 399
- TrackBox function 529 to 530
- TrackGoAway function 531 to 532
- transparent mode 609, 610
- 24-bit addressing
 - defined 247

U

- UncaptureComponent function 104
- unlocking relocatable blocks 252 to 253, 296 to 297, 359 to 360
- UnlockPixels procedure 719, 745 to 746
- UnregisterComponent function 90
- unregister request 52
- update events
 - routines for handling 534 to 535
- update events, and Standard File Package routines 204
- UpdateGWorld function 721, 735 to 738
- updating windows, saving memory space for 276
- user interface
 - for saving and opening files 177
- user interface guidelines 187
 - for highlighting 614
- user state of a window 490, 685
- useTempMem flag 725, 726, 730, 732
- useTextEditServices flag 474

V

- ValidRect routine 536 to 537
- ValidRgn routine 537
- variation codes
 - for windows 540, 563
- version 1 format 755 to 756
- version 2 format 755 to 756, 788
- version request 50 to 51
- video devices 679 to 713
- virtual memory
 - introduced 247
- visible regions
 - in color graphics ports 621
- volume control blocks 124
- volume reference numbers 124
- volumes
 - current 207 to 208
 - defined 120

- mounting 124
- naming 124
- offline 125
- selecting 184 to 187, 213 to 215

W

- WaitMouseUp function 454 to 455
- WaitNextEvent function 423 to 426
- WaitNextEvent function, and temporary memory 322
- 'wctb' resource type 491, 571
- 'WDEF' resource type 563, 571
- wedges
 - filling
 - with pixel patterns 646
- WinCTab data type 491 to 492
- window color table 490 to 492
- window definition functions
 - writing 563 to 567
- window definition IDs
 - and window definition functions 563
 - creating windows, used in 476, 508
 - in window resources 569
- window frames 564 to 565
- window list 494
- WindowList global variable 494
- Window Manager
 - application-defined routine for 562 to 567
 - data structures in 475 to 494
 - global variables 498
 - and pictures 763
 - port 497
 - resources in 567
 - routines in 497 to 551
 - initializing 497 to 498
 - low-level routines 546 to 551
- WindowPeek data type 484
- WindowPtr data type 484
- WindowRecord data type 484, 488 to 489
- window records 483 to 489
- window resources 568 to 571
- windows
 - closing 531 to 534
 - color in 490 to 493
 - creating 498 to 509
 - deallocating 532 to 534
 - displaying 511 to 517
 - grow image 512
 - hiding 513
 - maintaining update region of 534 to 537
 - manipulating
 - characteristics 537 to 540
 - color 544 to 546

- on the desktop 540 to 544
- moving 518 to 526
- naming 509 to 511
- retrieving information 517 to 518
- showing 513
- sizing 526 to 528
- standard state 490, 686
- user state 490, 685
- zooming 528 to 530, 685 to 688
- window state data record 489 to 490
- 'WIND' resource type 568 to 571
- WITH statement (Pascal), and dangling pointers 261
- working directory reference numbers 129
- writing data to files 137 to 140, 159
- WStateData data type 489 to 490

Z

- zero (memory location)
 - See 0 (memory location)
- zero-length handles
 - See 0-length handles
- Zone data structure 333
- zone headers 317, 332 to 334
- zone pointers 332
- zone records 332, 332 to 334
- zone trailer blocks 333
- zone trailers 317
- zooming windows 528 to 530, 685 to 688
- ZoomWindow procedure 686, 688
- ZoomWindow routine 530
- zzaComponent field 70
- zzaComponentInstance field 75, 94
- zzaddMax constant 609
- zzaddOver constant 608
- zzaddPin constant 608
- zzadMin constant 609
- zzalertPositionMainScreen constant 570
- zzalertPositionParentWindow constant 570
- zzalertPositionParentWindowScreen constant 570
- zzalignPix field 725, 727
- zzallDevices field 694 to 695
- zzallInit constant 693, 699, 707, 712
- zzallocPtr field 334
- zzaltDBoxProc constant 476
- zzappOpen constant 148
- zzappPrint constant 148
- zzarcCount field 781, 784
- zzarcProc field 630, 631
- zzawCTable field 493
- zzawFlags field 493
- zzawNext field 493
- zzawOwner field 493

- zzawRefCon field 493
- zzawReserved field 493
- zzbaseAddr field 616
- zzbitMapCount field 782, 784
- zzbitsProc field 630, 631
- zzbkColor field 620, 623
- zzbkLim field 333
- zzbkPixPat field 619, 621
- zzblend constant 608
- zzblue field 625, 627
- zzbounds field 616, 617
- zzburstDevice constant 693, 699, 707, 712
- zzcapturedComponent field 103
- zzcapturingComponent field 103
- zzcd field 76
- zzcenterMainScreen constant 570
- zzcenterParentWindow constant 570
- zzcenterParentWindowScreen constant 570
- zzCenterParetn constant 570
- zzchExtra field 619, 620
- zzci field 78
- zzclipPix field 726, 727
- zzclipRgn field 619
- zzclutType constant 692
- zzcmpCount field 616, 617
- zzcmpSize field 616, 618
- zzcmpWantsRegisterMessage constant 81
- zzcntEmpty field 333
- zzcntHandles field 333
- zzcntNRel field 333
- zzcntRel field 333
- zzcolorBankIs555 constant 811
- zzcolorBankIsCustom constant 811
- zzcolorBankIsExactAnd555 constant 811
- zzcolrBit field 620, 623
- zzcommentCount field 782, 784
- zzcommentHandle field 782, 784
- zzcommentProc field 630, 631
- zzcomponentEntryPoint field 86
- zzcomponent field 108
- zzcomponentFlags field 67, 81
- zzcomponentFlagsMask field 67, 82
- zzcomponentIcon field 76, 87
- zzcomponentInfo field 76, 87, 109
- zzcomponentManufacturer field 67, 81
- zzcomponentName field 76, 87, 109
- zzcomponentSubType field 66, 81
- zzcomponentType field 66, 81
- zzcontrRgn field 486
- zzcontrolList field 487
- zzcopy field 219
- zzcount field 779
- zzcr field 88
- zzctFlags field 626, 627
- zzctSeed field 626

- zzctSize field 491
- zzctSize field 626, 627
- zzctTable field 491
- zzctTable field 626, 627
- zzdataHandle field 487
- zzdBoxProc constant 476
- zzdefaultComponentAnyFlags flag 106
- zzdefaultComponentAnyManufacturer flag 106
- zzdefaultComponentAnySubType flag 106
- zzdefaultComponentIdentical flag 106
- zzdepth field 781, 783
- zzdevice field 619, 620
- zzdialogCItem field 493
- zzdialogKind constant 482
- zzdirectType constant 692
- zzditherCopy constant 607
- zzditherPix field 726, 727
- zzdocumentProc constant 476
- zzdontMatchSeeds field 694 to 695
- zzext32Device constant 693, 699, 707, 712
- zzfgColor field 619, 623
- zzfillPixPat field 619, 622
- zzfixedType constant 692
- zzflags field 83, 106
- zzflags field 333
- zzfName field 155, 219
- zzfontHandle field 782, 785
- zzfontNamesHandle field 782, 785
- zzfsAtMark constant 161
- zzfsCurPerm constant 135
- zzfsFromLEOF constant 161
- zzfsFromMark constant 161
- zzfsFromStart constant 161
- zzfsRdPerm constant 135
- zzfsRdWrPerm constant 135
- zzfsRdWrShPerm constant 135
- zzfsWrPerm constant 135
- zzftnNumber field 79
- zzfType field 155, 219
- zzfunc field 91
- zzgdCCBytes field 692, 694
- zzgdCCDepth field 692, 694
- zzgdCCXData field 692, 694
- zzgdCCXMask field 692, 694
- zzgdCompProc field 692, 693
- zzgdDevType constant 693, 699, 707, 709, 710, 712
- zzgdFlags field 692, 693
- zzgdID field 691, 692
- zzgdITable field 691, 692
- zzgdMode field 692, 694, 709
- zzgdNextGD field 692, 693
- zzgdPMap field 692, 693
- zzgdRect field 692, 693
- zzgdRefCon field 692, 693
- zzgdRefNum field 691, 692
- zzgdReserved field 692, 694
- zzgdResPref field 691, 692
- zzgdSearchProc field 692
- zzgdType field 691, 692
- zzgestalt32BitQD11 constant 588
- zzgestalt32BitQD12 constant 588
- zzgestalt32BitQD13 constant 588
- zzgestalt32BitQD constant 588
- zzgestalt8BitQD constant 588
- zzgestaltComponentMgr constant 35
- zzgestaltHasColor constant 589
- zzgestaltHasDeepGWorlds constant 589
- zzgestaltHasDirectPixMaps constant 589
- zzgestaltHasGrayishTextOr constant 589
- zzgetPicProc field 630, 631
- zzglobal field 86
- zzgoAwayFlag field 486
- zzgood field 219
- zzgrafProcs field 620, 624
- zzgrafVars field 619, 620
- zzgreen field 625, 627
- zzgwFlagErr field 726
- zzgWorldFlag4 field 725
- zzgWorldFlag5 field 725
- zzgWorldFlag8 field 725
- zzgWorldFlag9 field 725
- zzgWorldFlag10 field 725
- zzgWorldFlag11 field 725
- zzgWorldFlag12 field 725
- zzgWorldFlag13 field 725
- zzgWorldFlag14 field 725
- zzgWorldFlag15 field 725
- zzgWorldFlag21 field 726
- zzgWorldFlag22 field 726
- zzgWorldFlag23 field 726
- zzgWorldFlag24 field 726
- zzgWorldFlag25 field 726
- zzgWorldFlag26 field 726
- zzgWorldFlag27 field 726
- zzgzProc field 333
- zzhAxisOnly constant 521
- zzheapData field 334
- zzhFstFree field 333
- zzHighLevelEventMsgHeaderLength field 419
- zzhilite constant 614
- zzhilited field 486
- zzhRes field 616, 617, 778, 781, 783
- zzID field 779
- zzinContent constant 481
- zzinDrag constant 481
- zzinGoAway constant 481
- zzinGrow constant 481
- zzinMenuBar constant 480
- zzinZoomIn constant 481
- zzinZoomOut constant 481

zzkComponentCanDoSelect constant 83
 zzkComponentCloseSelect constant 83
 zzkComponentOpenSelect constant 42
 zzkComponentRegisterSelect constant 83
 zzkComponentSetTargetSelect constant 83
 zzkComponentUnregisterSelect constant 83
 zzkComponentVersionSelect constant 83
 zzkeepLocal field 725, 726
 zzlineCount field 781, 783
 zzlineProc field 630, 631
 zzlocation field 418, 420
 zzlooking field 70
 zzmainScreen constant 693, 699, 707, 712
 zzmapPix field 725, 727
 zzmatchData field 627
 zzmaxNRel field 333
 zzmaxRel field 333
 zzmedianMethod constant 797, 800, 803
 zzmessage field 414
 zzminCBFree field 333
 zzmodifiers field 416
 zzmoreMast field 333
 zzmovableDBoxProc constant 476
 zzmsgLength field 419
 zzname field 153, 417
 zznameOffset field 780
 zznewDepth field 725, 726
 zznewProc1 field 630
 zznewProc2 field 630
 zznewProc3 field 631
 zznewProc4 field 631
 zznewProc5 field 631
 zznewProc6 field 631
 zznewRowBytes field 725, 727
 zznextWindow field 487
 zznoAutoCenter constant 570
 zznoDriver constant 693, 699, 707, 712
 zznoGrtowDocProc constant 476
 zznoNewDevice field 725, 726
 zzopcodeProc field 630
 zzoriginalParams field 101
 zzovalCount field 781, 784
 zzovalProc field 630, 631
 zzpackSize field 616, 617
 zzpackType field 616, 617
 zzparams field 83
 zzparamSize field 83
 zzparID field 153
 zzpat1Data field 628
 zzpatData field 628
 zzpatMap field 628
 zzpatStretch field 620, 623
 zzpatType field 628, 629 to 630
 zzpatXData field 628
 zzpatXMap field 628
 zzpatXValid field 628
 zzpHiliteBit constant 612
 zzpicFrame field 777
 zzpicSave field 620, 624
 zzpicSize field 777
 zzpictFontID field 780
 zzpixelSize field 616, 617
 zzpixelsLocked field 725, 727
 zzpixelsPurgeable field 725, 726
 zzpixelType field 616, 617
 zzpixMapCount field 782, 784
 zzpixPurge field 725, 726
 zzplainDBox constant 476
 zzplaneBytes field 616, 618
 zzpmBkColor field 632
 zzpmBkIndex field 632
 zzpmFgColor field 632
 zzpmFgIndex field 632
 zzpmFlags field 632
 zzpmReserved field 616, 618
 zzpmTable field 616, 618
 zzpmVersion field 616, 617
 zzpnLoc field 619, 622
 zzpnLocHFrac field 619, 620
 zzpnMode field 619, 622
 zzpnPixPat field 619, 622
 zzpnSize field 619, 622
 zzpnVis field 619, 622
 zzpolyCount field 782, 784
 zzpolyProc field 630, 631
 zzpolySave field 620, 624
 zzpopularMethod constant 797, 800, 803
 zzport field 485
 zzportPixMap field 619, 620
 zzportRect field 619
 zzportVersion field 619, 620
 zzpostingOptions field 419
 zzpurgeProc field 334
 zzpurgePtr field 333
 zzputPicProc field 630, 631
 zzramInit constant 693, 699, 707, 712
 zzrDocProc constant 476
 zzreallocPix field 726, 727
 zzrecordComments constant 797, 802
 zzrecordFontInfo constant 797, 802
 zzrectCount field 781, 784
 zzrectProc field 630, 631
 zzrecvrName field 418, 420
 zzred field 625, 627
 zzrefCon field 487
 zzrefCon field 201
 zzrefnum field 100
 zzregionCount field 782, 784
 zzregisterCmpGlobal constant 89
 zzregisterCmpGlobal flag 86

zzregisterCmpNoDuplicates constant 89
 zzregisterCompAfter constant 89
 zzregisterCompAfter flag 87
 zzreserved1 field 419
 zzresId field 109
 zzresRefNum field 89
 zzresType field 109
 zzreturnColorTable constant 797, 800, 802
 zzreturnPalette constant 797, 800, 802
 zzrgbBkColor field 619, 621
 zzRGBDirect constant 617
 zzrgbFgColor field 619, 621
 zzrgb field 491
 zzrgb field 625, 626
 zzrgbHiliteColor field 632
 zzrgbOpColor field 632
 zzrgnProc field 630, 631
 zzrgnSave field 620, 624
 zzrowBytes field 616, 617
 zzrRectCount field 781, 784
 zzrRectProc field 630, 631
 zzscreenActive constant 693, 699, 707, 712
 zzscreenDevice constant 693, 699, 707, 712
 zzsessionId field 417
 zzsfErrorDialogRefCon constant 202
 zzsfFile field 154, 189, 218
 zzsfFlags field 155, 218
 zzsfGood field 154, 189, 217
 zzsfHookChangeSelection constant 200
 zzsfHookCharOffset constant 200
 zzsfHookFirstCall constant 199
 zzsfHookFolderPopUp constant 200
 zzsfHookGoToAliasTarget constant 201
 zzsfHookGoToDesktop constant 201
 zzsfHookGoToNextDrive constant 201
 zzsfHookGoToParent constant 201
 zzsfHookGoToPrevDrive constant 201
 zzsfHookLastCall constant 199
 zzsfHookNullEvent constant 199
 zzsfHookOpenAlias constant 200
 zzsfHookOpenFolder constant 200
 zzsfHookRebuildList constant 200
 zzsfHookSetActiveOffset constant 200
 zzsfIsFolder field 155, 218
 zzsfIsVolume field 155, 218
 zzsfItemBalloonHelp constant 197
 zzsfItemCancelButton constant 198
 zzsfItemDesktopButton constant 199
 zzsfItemDividerLinePict constant 197
 zzsfItemEjectButton constant 198
 zzsfItemFileListUser constant 199
 zzsfItemFileNameTextEdit constant 199
 zzsfItemNewFolderUser constant 199
 zzsfItemOpenButton constant 198
 zzsfItemPopUpMenuUser constant 199
 zzsfItemPromptStaticText constant 197
 zzsfItemVolumeUser constant 198
 zzsfLockWarnDialogRefCon constant 202
 zzsfMainDialogRefCon constant 201
 zzsfNewFolderDialogRefCon constant 201
 zzsfReplaceDialogRefCon constant 202
 zzsfReplacing field 154, 189, 217
 zzsfReserved1 field 155, 218
 zzsfReserved2 field 155, 218
 zzsfScript field 154, 218
 zzsfStatWarnDialogRefCon constant 202
 zzsfType field 154, 217
 zzsingleDevices field 694 to 695
 zzsize field 780
 zzsmSystemScript constant 154, 218
 zzsourceRect field 781, 783
 zzspareFlag field 486
 zzsparePtr field 334
 zzspExtra field 619, 623
 zzsrcRect field 778
 zzstaggerMainScreen constant 570
 zzstaggerParentWindow constant 570
 zzstaggerParentWindowScreen constant 571
 zzstdState field 490
 zzstorage field 92
 zzstretchPix field 726, 727
 zzstrucRgn field 486
 zzstyle field 780
 zzsubOver constant 609
 zzsubPin constant 608
 zzsuppressBlackAndWhite constant 797, 800, 802
 zzsysFontID field 780
 zzsystemMethod constant 797, 800, 803
 zztarget field 105
 zztextCount field 781, 783
 zztextProc field 630, 631
 zztheA5 field 96
 zztheColorTable field 781, 783
 zztheError field 97
 zztheMsgEvent field 419
 zzthePalette field 781, 782
 zztheRefCon field 98
 zztheStorage field 94
 zztitleHandle field 487
 zztitleWidth field 487
 zztransparent constant 609
 zztxFace field 619, 623
 zztxFont field 619, 623
 zztxMeasProc field 630, 631
 zztxMode field 619, 623
 zztxSize field 619, 623
 zzuniqueColors field 781, 782
 zzuniqueComments field 782, 784
 zzuniqueFonts field 782, 785
 zzupdateRgn field 486

zzuserKind constant 482
 zzuserRefCon field 419
 zzuserState field 489
 zzuseTempMem field 725, 726
 zzvalue field 491
 zzvalue field 625, 626
 zzvAxisOnly constant 521
 zzversion field 778, 781, 782
 zzversion field 219, 419
 zzversNum field 155
 zzvisible field 485
 zzvisRgn field 619
 zzvRefNum field 153, 155, 219
 zzvRes field 616, 617, 778, 781, 783
 zzwCalcRgns constant 566
 zzwContentColor constant 482
 zzwCReserved field 491
 zzwCSeed field 491
 zzwDialogDark constant 483
 zzwDialogLight constant 483
 zzwDispose constant 566
 zzwDraw constant 564
 zzwDrawGIcon constant 567
 zzwFrameColor constant 482
 zzwGrow constant 566
 zzwhat field 83
 zzwhat field 414
 zzwhen field 416
 zzwhere field 416
 zzwHiliteColor constant 482
 zzwHiliteColorDark constant 482
 zzwHiliteColorLight constant 482
 zzwHit constant 565
 zzwindowDefProc field 486
 zzwindowKind field 485
 zzwindowPic field 487, 533, 534, 538, 539
 zzwNew constant 566
 zzwTextColor constant 482
 zzwTingeDark constant 483
 zzwTingeLight constant 483
 zzwTitleBarColor constant 482
 zzwTitleBarDark constant 483
 zzwTitleBarLight constant 482
 zzzcbFree field 333
 zzzoomDocProc constant 476
 zzzoomNoGrow constant 476