

Plate 1

Device RGB color space and CMYK color space

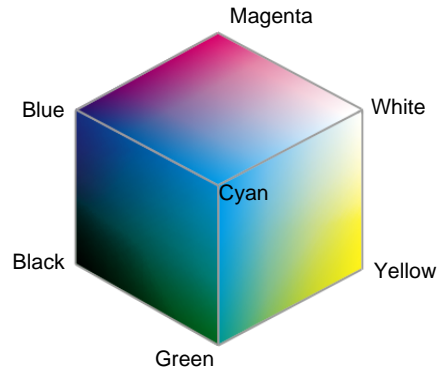


Plate 2

The HSV and HLS color space cones

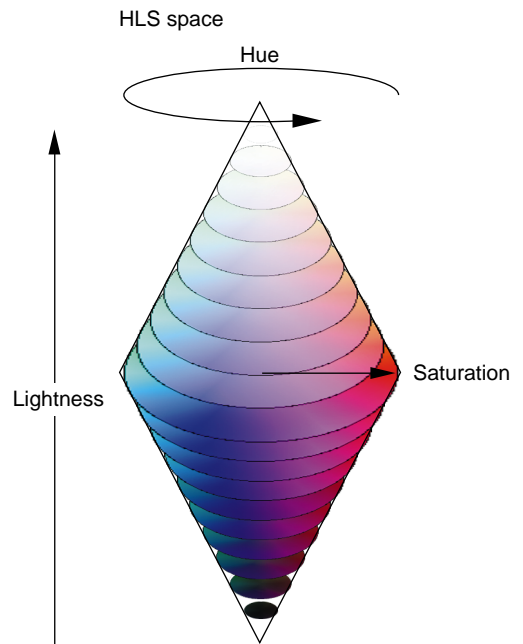
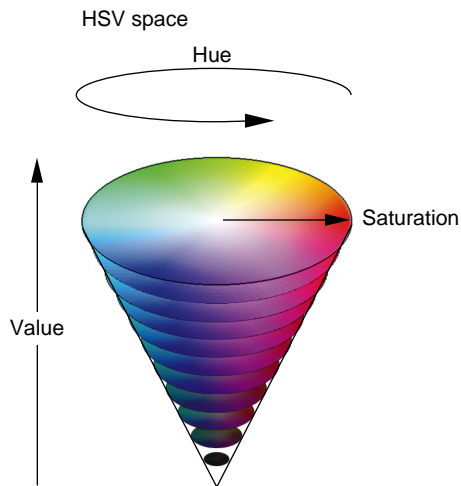
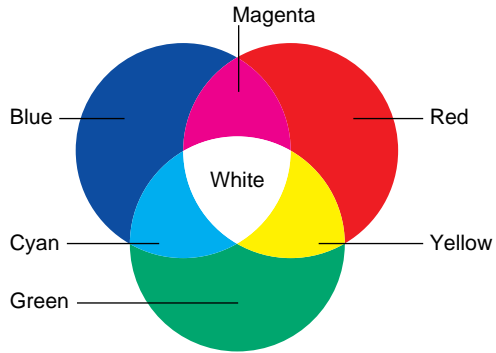


Plate 3

Additive and subtractive color

Additive color



Subtractive color

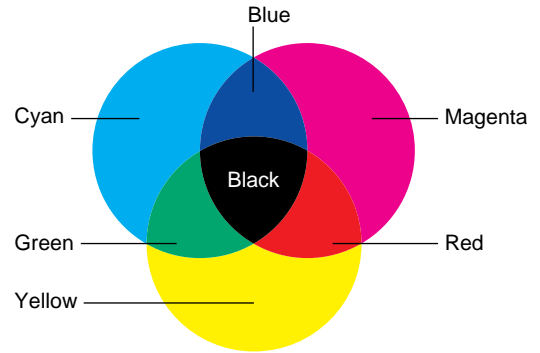


Plate 4

CIE color space

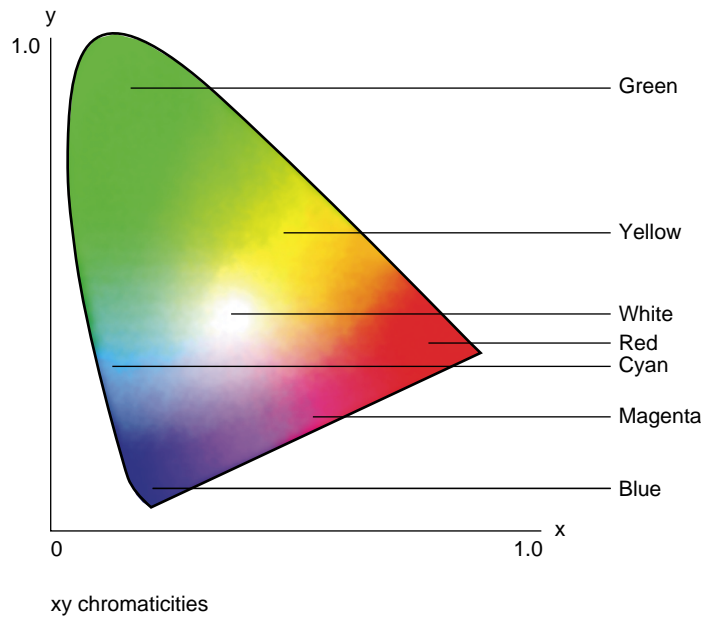


Plate 5 The default color picker

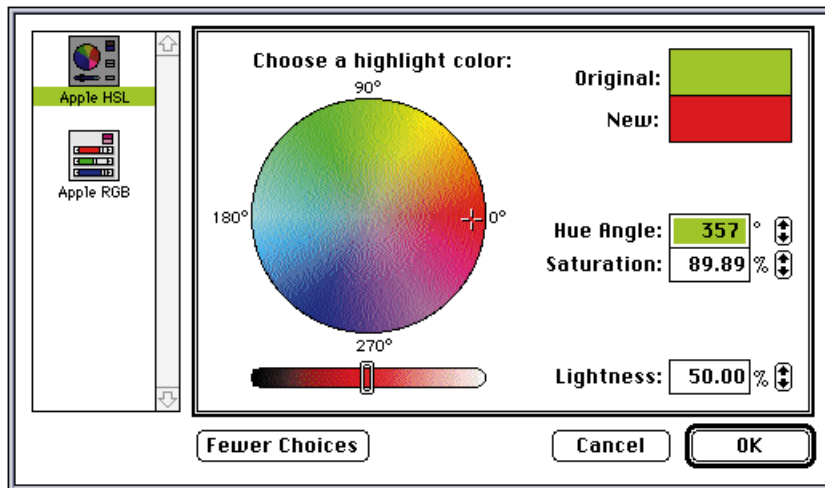


Plate 6 L*a*b* color space

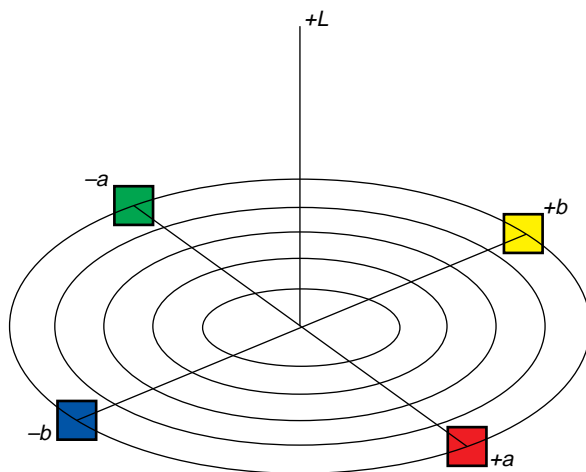



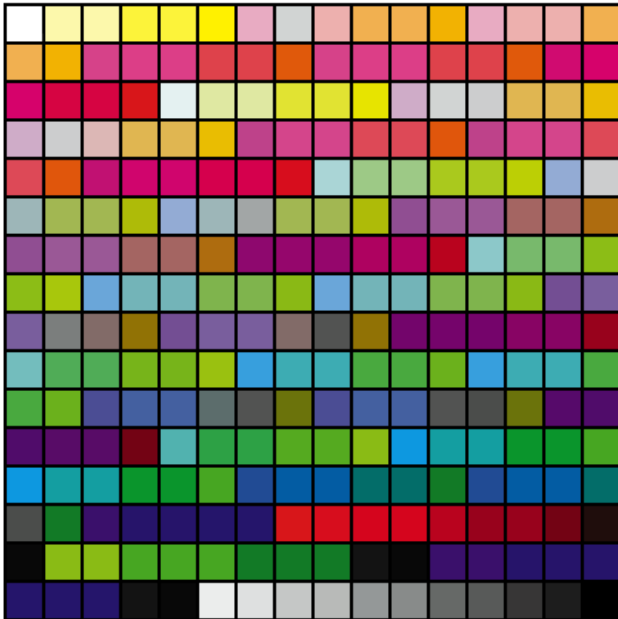
Plate 7

The colors of the default color tables

1 bit 

2 bits 

4 bits 

8 bits 

Advanced Color Imaging on the Mac OS



Addison-Wesley Publishing Company

Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan
Paris Seoul Milan Mexico City Taipei

Apple Computer, Inc.
© 1986, 1995 Apple Computer, Inc.
All rights reserved.

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

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

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

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

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

Balloon Help, Finder and QuickDraw, are trademarks of Apple Computer, Inc.

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

AGFA is a trademark of Agfa-Gevaert.

America Online is a registered service mark of America Online, Inc.

CompuServe is a registered service mark of CompuServe, Inc.

FrameMaker is a registered trademark of Frame Technology Corporation.

Helvetica and Palatino are registered trademarks of Linotype Company.

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

PowerPC™ and the PowerPC logo™ are trademarks of International Business Machines Corporation, used under license therefrom.

QuickView™ is licensed from Altura Software, Inc.

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY

(90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

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

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

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

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

ISBN 0-201-48949-X
1 2 3 4 5 6 7 8 9-MA-9998979695
First Printing, August 1995

Library of Congress Cataloging-in-Publication Data

Advanced color imaging on the Mac OS.

p. cm.

Includes index.

ISBN 0-201-48949-X (pbk.)

1. Color computer graphics. 2. Macintosh (Computer)

I. Apple Computer, Inc.

T385.A3598 1995

006.6'765--dc20

95-21221
CIP

Contents

Figures, Tables, and Listings xi

Preface About This Book xv

Format of This Book and Its Companion Volume xvi

Conventions Used in This Book xviii

 Special Fonts xviii

 Types of Notes xviii

Development Environment xix

For More Information xix

Chapter 1 Palette Manager 1-1

About the Palette Manager 1-4

 Palette Format 1-5

 The Palette Paradigm 1-5

 Colors in a Palette 1-6

 Courteous Colors 1-9

 Tolerant Colors 1-11

 Animated Colors 1-12

 Displaying Animated Colors on Direct Devices 1-13

 Explicit Colors 1-14

 Inhibited Colors 1-15

 Combining Color Usage for an Entry 1-16

 Sequencing the Entries 1-16

 How the Palette Manager Allocates Colors for Display 1-17

 How the Palette Manager Restores the Color Environment 1-18

Using the Palette Manager 1-20

 Creating Palettes 1-21

 Creating a Palette in Code 1-21

 Creating a Palette in a Resource File 1-23

 Selecting the Right Color Set 1-25

 Creating a Palette by Copying and Assigning It to a Window 1-27

Designating a Default Palette for Your Application	1-29
Drawing With a Palette's Colors	1-31
Animating a Window With a Palette	1-31
Disposing of a Palette and Restoring the Color Table	1-33
Using Palettes With Offscreen Graphics Worlds	1-34
Summary of the Palette Manager	1-38
Constants	1-38
Data Types	1-38
Functions	1-39

Chapter 2 **Color Picker Manager** 2-1

About the Color Picker Manager	2-3
Color Picker Dialog Boxes	2-4
Color Pickers as Components	2-6
ColorSync Colors and the Color Picker Manager	2-7
Using the Color Picker Manager	2-8
Using the Standard Dialog Box for Color Pickers	2-9
Defining an Event Filter Function	2-10
Defining a Color-Changed Function	2-11
Using Customized Dialog Boxes for Color Pickers	2-13
Creating Dialog Boxes for Color Pickers	2-14
Setting Colors for and Getting Colors From the Color Picker	2-19
Handling Events in a Color Picker Dialog Box	2-21
Handling Events in the Edit Menu	2-24
Sending Event Forecasters to the Color Picker	2-27
Setting the Destination Profile	2-28
Controlling the Help Balloons for a Color Picker's Dialog Box	2-29
Writing Your Own Color Pickers	2-31
Creating a Component Resource for a Color Picker	2-32
Dispatching to Functions Defined by a Color Picker	2-34
Initializing Your Color Picker	2-37
Handling Events for Your Color Picker	2-41
Returning and Setting Color Picker Information	2-44

Summary of the Color Picker Manager	2-51
Constants and Data Types	2-51
Color Picker Manager Functions	2-57
Application-Defined Functions	2-60
Color Picker-Defined Functions	2-60

Chapter 3 Introduction to the ColorSync Manager 3-1

Introduction to Color and Color Management Systems	3-4
Color: A Brief Overview	3-4
Color Perception	3-5
Hue, Saturation, and Brightness	3-5
Additive and Subtractive Color	3-6
Color Spaces	3-6
Gray Spaces	3-7
RGB-Based Color Spaces	3-7
CMY-Based Color Spaces	3-10
Device-Independent Color Spaces	3-11
Indexed Color Spaces	3-14
Color-Component Values, Color Values, and Colors	3-15
Color Conversion and Color Matching	3-15
Color Management Systems	3-17
About the ColorSync Manager	3-18
Programming Interfaces	3-18
About the ColorSync Manager's Memory Allocation and Use	3-19
Profiles	3-19
Color Management Modules	3-22
When Color Matching Occurs	3-24
QuickDraw GX and the ColorSync Manager	3-26
What Users Can Do With ColorSync-Supportive Applications	3-27
Display Matching	3-27
Gamut Checking	3-27
Soft Proofing	3-28
Device-Linked Profiles	3-28
Calibration	3-28

About ColorSync Application Development	4-4	
About the ColorSync Manager Programming Interface	4-4	
What Should a ColorSync-Supportive Application Do?	4-5	
At a Minimum	4-5	
Storing and Handling Profiles	4-6	
How the ColorSync Manager Selects the CMM to Be Used	4-7	
Developing Your ColorSync-Supportive Application	4-12	
Determining If the ColorSync Manager Is Available	4-14	
Providing Minimal ColorSync Support	4-15	
Obtaining Profile References	4-16	
Opening a Profile and Obtaining a Reference to It	4-17	
Identifying the Current System Profile	4-19	
Matching Colors to Displays Using ColorSync With QuickDraw Operations	4-20	
Matching Colors in a Picture Containing an Embedded Profile	4-21	
Matching Colors as Your User Draws a Picture	4-22	
Setting a Large Profile Element	4-24	
Creating a Color World for Color Matching and Checking Using the Low-Level Functions	4-27	
Matching Colors Using the Low-Level Functions	4-29	
Matching the Colors of a Pixel Map to the Display's Color Gamut	4-30	
Matching the Colors of a Bitmap Image to the Display's Color Gamut	4-31	
Embedding Profiles in Documents and Pictures	4-34	
Extracting Profiles Embedded in Pictures	4-38	
Step 1: Counting the Profiles in the PICT File	4-40	
Step 2: Extracting the Profile	4-42	
Searching for Profiles in the ColorSync™ Profiles Folder	4-49	
Checking Colors Against a Destination Device's Gamut	4-51	
Creating and Using Device-Linked Profiles	4-53	
Considerations	4-56	
Providing Soft Proofs	4-56	
Calibrating a Device	4-58	

Summary of the ColorSync Manager	4-59
Constants	4-59
Data Structures	4-63
Functions	4-69

Chapter 5 **Developing Color Management Modules** 5-1

About Color Management Modules	5-4
Creating a Color Management Module	5-6
Creating a Component Resource for a CMM	5-6
How Your CMM Is Called by the Component Manager	5-9
Handling Request Codes	5-10
Responding to Required Component Manager Request Codes	5-21
Responding to ColorSync Manager Required Request Codes	5-22
Responding to ColorSync Manager Optional Request Codes	5-25
Summary of the Color Management Modules	5-39
Constants	5-39
Functions	5-40

Chapter 6 **Developing ColorSync-Supportive Device Drivers** 6-1

About ColorSync-Supportive Device Driver Development	6-4
The ColorSync Manager Programming Interface	6-4
Devices and Their Profiles	6-5
The Profile Format and Its Cross-Platform Use	6-6
Storing and Handling Device Profiles	6-6
How You Use Profiles	6-7
Devices and Color Management Modules	6-8
Providing ColorSync-Supportive Device Drivers	6-8
Providing Minimum Support	6-9
Providing More Extensive ColorSync Support	6-9
Developing Your ColorSync Supportive Device Driver	6-10
Determining If the ColorSync Manager Is Available	6-11
Interacting With the User	6-11
Searching for Profiles and Displaying Their Names to the User	6-12

Setting the Rendering Intent Selected by the User	6-15
Setting the Color-Matching Quality Selected by the User	6-17
Color Matching an Image to Be Printed	6-22

Chapter 7 **Color Manager** 7-1

About the Color Manager	7-3
Graphics Devices	7-4
Color Tables	7-5
Inverse Tables	7-6
Inverse Tables in Action	7-10
Hidden Colors	7-11
Building Inverse Tables	7-12
Using the Color Manager	7-13
Customizing Search Functions	7-13
Customizing Complement Functions	7-15
Managing the Device CLUT	7-16
Summary of the Color Manager	7-19
Constants and Data Types	7-19
Color Manager Functions	7-20
Application-Defined Functions	7-21

Appendix **ColorSync Manager Backward Compatibility** A-1

ColorSync 1.0 API Support	A-1
ColorSync 1.0 Profile Support	A-2
ColorSync 1.0 Profiles and Version 2.0 Profiles	A-2
How ColorSync 1.0 Profiles and Version 2.0 Profiles Differ	A-3
CMMs and Mixed Profiles	A-5
Using the ColorSync Manager API With ColorSync 1.0 Profiles	A-5
ColorSync Manager Functions Not Supported for ColorSync 1.0 Profiles	A-6
Using ColorSync 1.0 Profiles With the ColorSync Manager	A-7
Opening a ColorSync 1.0 Profile	A-7
Obtaining a ColorSync 1.0 Profile Header	A-7

Obtaining ColorSync 1.0 Profile Elements	A-8
Embedding ColorSync 1.0 Profiles	A-8
ColorSync 1.0 Functions With Parallel ColorSync Manager Counterparts	A-9

Glossary	GL-1
----------	------

Index	IN-1
-------	------

Figures, Tables, and Listings

Color Plates

Color plates are immediately preceding the title page.

Color Plate 1	Device RGB color space and CMYK color space
Color Plate 2	The HSV and HLS color space cones
Color Plate 3	Additive and subtractive color
Color Plate 4	CIE color space
Color Plate 5	The default color picker
Color Plate 6	L*a*b* color space
Color Plate 7	The colors of the default color tables

Preface	About This Book	xv
---------	-----------------	----

Figure P-1	Road Map to <i>Advanced Color Imaging</i>	xvii
-------------------	---	------

Chapter 1	Palette Manager	1-1
-----------	-----------------	-----

Table 1-1	A courteous palette	1-10
Listing 1-1	Creating a red palette	1-22
Listing 1-2	A palette ('pltt') resource	1-23
Listing 1-3	Displaying different colors on different types of screens	1-25
Listing 1-4	Copying a color table to a palette	1-28
Listing 1-5	Animating with a palette	1-32
Listing 1-6	Changing the color environment of an offscreen graphics world	1-35

Chapter 2	Color Picker Manager	2-1
-----------	----------------------	-----

Figure 2-1	The standard dialog box for color pickers	2-4
Figure 2-2	Color picker choices in the standard dialog box for color pickers	2-5
Figure 2-3	A movable modal dialog box for color pickers	2-6
Figure 2-4	A movable modal application-owned dialog box	2-17
Figure 2-5	An application-created color picker	2-33

Listing 2-1	Using the <code>PickColor</code> function	2-9
Listing 2-2	An event filter function for the <code>PickColor</code> function	2-11
Listing 2-3	A color-changed function	2-12
Listing 2-4	Creating a movable modal system-owned dialog box	2-15
Listing 2-5	Creating an application-owned dialog box	2-16
Listing 2-6	Creating a color picker–owned dialog box	2-18
Listing 2-7	Setting the original and new colors	2-19
Listing 2-8	Determining the selected color	2-20
Listing 2-9	A sample event loop	2-22
Listing 2-10	Handling the Edit menu	2-25
Listing 2-11	Warning the color picker that it's about to be closed	2-28
Listing 2-12	Using the <code>SetPickerProfile</code> function to set the destination profile	2-29
Listing 2-13	Using the <code>GetPickerProfile</code> function to get the destination profile	2-29
Listing 2-14	Using the <code>ExtractPickerHelpItem</code> function	2-30
Listing 2-15	A component resource for a color picker	2-32
Listing 2-16	Handling Component Manager request codes	2-36
Listing 2-17	Initializing private data for a color picker	2-38
Listing 2-18	Testing whether an environment can support your color picker	2-39
Listing 2-19	Returning the dialog box items for a color picker	2-40
Listing 2-20	Redrawing a color picker	2-41
Listing 2-21	Responding to events before handing them to the Dialog Manager	2-41
Listing 2-22	Responding to events in color picker items	2-42
Listing 2-23	Handling events in the color picker's Edit menu	2-44
Listing 2-24	Returning the original or the new color	2-45
Listing 2-25	Setting colors	2-45
Listing 2-26	Returning icon data	2-47
Listing 2-27	Returning the color picker's prompt	2-48
Listing 2-28	Setting the color picker's prompt	2-48
Listing 2-29	Returning the destination profile	2-49
Listing 2-30	Setting the destination profile	2-49
Listing 2-31	Specifying how the Edit menu should be set	2-50

Chapter 3	Introduction to the ColorSync Manager	3-1
Figure 3-1	Gray space	3-7
Figure 3-2	RGB color space	3-8
Figure 3-3	HSV color space and HLS color space	3-9
Figure 3-4	Additive colors expressed in CMYK and subtractive colors expressed in RGB	3-10
Figure 3-5	Yxy chromaticities in the CIE color space	3-13
Figure 3-6	Color gamuts for two devices expressed in Yxy space	3-16
Figure 3-7	The ColorSync™ System Profile control panel	3-25
Figure 3-8	The ColorSync Manager and the Component Manager	3-26
Chapter 4	Developing ColorSync-Supportive Applications	4-1
Figure 4-1	Color matching when the source and destination profiles specify the same CMM	4-7
Figure 4-2	Color matching using the destination profile's CMM	4-8
Figure 4-3	Color matching using the source profile's CMM	4-9
Figure 4-4	Color matching through an XYZ interchange space using both CMMs	4-10
Figure 4-5	Matching using both CMMs and two interchange color spaces	4-11
Figure 4-6	Color matching using the Apple-supplied default CMM	4-12
Listing 4-1	Opening a reference to a file-based profile	4-18
Listing 4-2	Obtaining the current system profile	4-20
Listing 4-3	Two methods of color matching to a display	4-23
Listing 4-4	Setting the element size before setting the element data in segments	4-24
Listing 4-5	Matching the colors of a pixel map or a bitmap using a color world	4-32
Listing 4-6	Embedding a profile by prepending it before its associated picture	4-36
Listing 4-7	Counting the number of profiles in a picture	4-41
Listing 4-8	Calling the CMUnflattenProfile function to extract an embedded profile	4-43
Listing 4-9	The unflatten procedure	4-45
Listing 4-10	The comment procedure	4-47
Listing 4-11	Searching for specific profiles in the ColorSync™ Profiles folder	4-50

Chapter 5	Developing Color Management Modules	5-1
	Figure 5-1	The ColorSync Manager and the Component Manager 5-5
	Listing 5-1	CMM component Rez listing 5-8
	Listing 5-2	A CMM component shell 5-14
Chapter 6	Developing ColorSync-Supportive Device Drivers	6-1
	Listing 6-1	Modifying the system profile header's quality flag and setting the header 6-20
Chapter 7	Color Manager	7-1
	Figure 7-1	Sample inverse table 7-7
	Figure 7-2	An inverse table of resolution 4 7-9
	Figure 7-3	Creating an inverse table index 7-11
	Table 7-1	Sample inverse table indexes 7-10
	Table 7-2	A sample <code>CSpecArray</code> data structure 7-17
	Listing 7-1	Adding and using a custom search function 7-14
Appendix	ColorSync Manager Backward Compatibility	A-1
	Table A-1	ColorSync 1.0 functions and their ColorSync Manager counterparts A-9

About This Book

This book, *Advanced Color Imaging on the Mac OS*, and its (electronic) companion, *Advanced Color Imaging Reference*, describe the following collections of system software routines:

- the Palette Manager
- the Color Picker Manager, version 2.0
- the ColorSync Manager, version 2.0
- the Color Manager

The chapters in this book describe how to use these managers to enhance your application's color capabilities. To implement core graphics capabilities, your application should use QuickDraw or QuickDraw GX. The book *Inside Macintosh: Imaging With QuickDraw* describes how your application can use QuickDraw to create and display Macintosh graphics, and how to use the Printing Manager to print the images created with QuickDraw. The *Inside Macintosh: QuickDraw GX* suite of books describes the QuickDraw GX object-based graphics programming environment for creating, displaying, and printing graphics.

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

To solicit color choices from users, your application can use the Color Picker Manager. Whether your application uses QuickDraw or QuickDraw GX, the Color Picker Manager provides your application with a standard dialog box for soliciting a color choice from users.

To match colors between screens and input and output devices such as scanners and printers, Macintosh system software provides a set of routines and algorithms called the ColorSync Manager. Developers writing device drivers use the ColorSync Manager to support color matching between devices.

Application developers use the ColorSync Manager to communicate with drivers and to present users with color-matching information—such as a device’s color capabilities.

QuickDraw GX and the Color Picker Manager automatically use the ColorSync Manager to perform color matching. Unless your application is using one of these two graphics managers, it must explicitly call the functions of the ColorSync Manager to use its color-matching capabilities.

The Color Manager assists Color QuickDraw in mapping your application’s color requests to the actual colors available. Most applications never need to call the Color Manager directly.

Format of This Book and Its Companion Volume

This book provides conceptual information about enhancing your application’s color capabilities; it also includes code samples that give step-by-step instructions for doing so. For example, in the chapter “Color Picker Manager,” conceptual information is in the section “About the Color Picker Manager,” which explains how you can use the standard user interface for soliciting color choices from users.

Tutorial information is in the section “Using the Color Picker Manager,” which contains code samples and step-by-step instructions describing how to use the Color Picker Manager to create dialog boxes in which users can make color choices. The chapter “Color Picker Manager” also contains a summary section that provides the C interfaces for the constants, data structures and functions associated with the Color Picker Manager.

The electronic companion to this book, *Advanced Color Imaging Reference*, provides a reference chapter for each of the managers described in *Advanced Color Imaging on the Mac OS*. For example, the chapter “Color Picker Manager Reference” provides a complete reference to the data structures, functions, and resources that your application can use to create an interface for soliciting color choices from users. Each function description also follows a standard format, which presents the function definition followed by a description of every parameter of the function.

The book *Advanced Color Imaging on the Mac OS* is in a printed form and an electronic form. The content of these two versions is identical.





The *Advanced Color Imaging Reference* is in an electronic form only—there is no printed version of it. It has two electronic formats that are identical in content:

- **Adobe™ Acrobat™ format.** Acrobat features excellent navigation and the ability to print the entire document or selected pages.
- **QuickView format.** QuickView features extremely fast navigation and limited printing capabilities. For better printing capabilities, it is suggested that you use the Adobe Acrobat version.

For additional information on navigating in the *Advanced Color Imaging Reference* with Acrobat or QuickView, see the ReadMe file on the enclosed CD.

Figure P-1 shows a road map to the printed and electronic documentation forms of *Advanced Color Imaging on the Mac OS* and the *Advanced Color Imaging Reference*.

Figure P-1 Road Map to *Advanced Color Imaging*

Location → Content ↓	Paper documentation	Electronic documentation	
		Acrobat format	QuickView format
Conceptual, introductory, and tutorial information	 <i>Advanced Color Imaging on the Mac OS</i>	 <i>Advanced Color Imaging on the Mac OS</i>	
Reference		 <i>Advanced Color Imaging Reference</i>	 <i>Advanced Color Imaging Reference</i>

Conventions Used in This Book

This book uses various conventions to present 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 Letter Gothic (`this is Letter Gothic`).

Words that appear in **boldface** are key terms or concepts and are defined in the glossary at the end of this book.

Types of Notes

There are several types of notes used in this book.

Note

A note like this contains information that is interesting but possibly not essential to an understanding of the main text. (An example appears on page 1-21.) ♦

IMPORTANT

A note like this contains information that is essential for an understanding of the main text. (An example appears on page 1-30.) ▲

▲ **WARNING**

Warnings like this indicate potential problems that you should be aware of as you design your application. Failure to heed these warnings could result in system crashes or loss of data. (An example appears on page 1-14.) ▲

Development Environment

The system software functions described in this book are available using C or assembly-language interfaces. How you access these functions depends on the development environment you are using. This book shows system software routines in their C interface using the Macintosh Programmer's Workshop (MPW).

All code listings in this book are shown in C (except for listings that describe resources, which are shown in Rez-input format). They show methods of using various routines and illustrate techniques for accomplishing particular tasks. All code listings have been compiled and, in most cases, tested. However, Apple Computer does not intend that you use these code samples in your application. You can find the location of this book's code listings in the list of figures, tables, and listings.

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

For More Information

APDA is Apple Computer's worldwide source for hundreds of development tools, technical resources, training products, and information for anyone interested in developing applications on Apple platforms. Customers receive the *APDA Tools Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. APDA offers convenient payment and shipping options, including site licensing.

P R E F A C E

To order products or to request a complimentary copy of the *APDA Tools Catalog*, contact

APDA
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone	1-800-282-2732 (United States) 1-800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
AppleLink	APDA
America Online	APDAorder
CompuServe	76666,2405
Internet	APDA@applelink.apple.com

If you provide commercial products and services, call 408-974-4897 for information on the developer support programs available from Apple.

For information on registering signatures, file types, and other technical information, contact

Macintosh Developer Technical Support
Apple Computer, Inc.
20525 Mariani Avenue, M/S 303-2T
Cupertino, CA 95014-6299

Palette Manager

Contents

About the Palette Manager	1-4
Palette Format	1-5
The Palette Paradigm	1-5
Colors in a Palette	1-6
Courteous Colors	1-9
Tolerant Colors	1-11
Animated Colors	1-12
Displaying Animated Colors on Direct Devices	1-13
Explicit Colors	1-14
Inhibited Colors	1-15
Combining Color Usage for an Entry	1-16
Sequencing the Entries	1-16
How the Palette Manager Allocates Colors for Display	1-17
How the Palette Manager Restores the Color Environment	1-18
Using the Palette Manager	1-20
Creating Palettes	1-21
Creating a Palette in Code	1-21
Creating a Palette in a Resource File	1-23
Selecting the Right Color Set	1-25
Creating a Palette by Copying and Assigning It to a Window	1-27
Designating a Default Palette for Your Application	1-29
Drawing With a Palette's Colors	1-31
Animating a Window With a Palette	1-31
Disposing of a Palette and Restoring the Color Table	1-33
Using Palettes With Offscreen Graphics Worlds	1-34

Summary of the Palette Manager	1-38
Constants	1-38
Data Types	1-38
Functions	1-39

This chapter describes how you can use the Palette Manager to ensure that an optimal set of colors is available whenever one of your application's windows is active.

The Palette Manager monitors the color needs of the graphics environment. The Palette Manager can track the combined color and grayscale requirements of your application, other applications, and the Operating System, and it can do so across multiple screens. The primary purpose of the Palette Manager is to ensure that the best set of colors is available for display devices with limited color capabilities.

For example, on a device that can display 16 colors only, an image composed entirely of earth tones will lack all subtlety and richness because the default color lookup table for the device provides a broad set of colors spread across the color spectrum. With the Palette Manager you can create a palette containing a range of 14 earth tones (you must provide black and white as well) and attach it to a window to get the best display possible for your image. This palette can be attached to one particular window and needn't affect the color display of the Finder, of other applications, or of other windows in your application.

You need to read this chapter if your application uses Color QuickDraw's color system, rather than the basic eight-color system supplied with original QuickDraw and you need to control your color environment.

You should be familiar with the material in *Inside Macintosh: Imaging With QuickDraw*. In particular, you should understand how Color QuickDraw and graphic devices such as video cards display colors and grays on a screen. The chapter "Introduction to QuickDraw" in *Inside Macintosh: Imaging With QuickDraw* describes the format and uses of color lookup tables, and it introduces both the indexed color system that can display up to 256 colors and the direct color system that can display thousands or millions of colors. In the same book, the chapter "Color QuickDraw" provides details about the color graphics port and the `grafVars` data structure, which together hold information about the color settings.

Because the Palette Manager uses the Color Manager to coordinate color assignments across multiple graphic devices, most developers don't need to read the Color Manager chapter, which itself operates on single devices only.

About the Palette Manager

You can use the Palette Manager to

- create a palette so that an optimal set of colors and grays is available to your application's window
- provide sets of colors for displaying images on devices with different pixel depths
- create color animation effects

Indexed devices (devices with a pixel depth of 8 bits or less) have default color lookup tables (CLUTs) that contain a broad spectrum of colors to meet the color needs of the widest range of applications. The Palette Manager allows you to customize the colors for each window in your application—within the limits of each display device's pixel depth. The Palette Manager can enhance your color display because it can give you a specialized range of colors to display. For example, on a device with an 8-bit pixel depth, instead of 256 colors spread across the spectrum, which is what the default CLUT provides, you could choose a particular range of colors depending on the image your application is displaying in a specific window. For one image you might provide 180 greens and the remaining 76 colors across the spectrum, or for a different image, 254 reds with black and white.

The Palette Manager compares your palettes with the CLUTs for the available display devices. For example, if a user opens your application's window or makes it active and frontmost on the desktop, the Palette Manager compares the colors of the palette you specified for your window with those available for the video card and loads whatever colors are needed into the color table of the card.

On direct devices, the Palette Manager uses the exact colors you request, to the limits of the card's pixel depth (15 or 24 bits per pixel). The use of palettes does not enhance the color display on direct devices because direct devices can display thousands or millions of colors, which means that sufficient colors are available for even the most skewed or narrow color scheme.

Palette Format

A **palette** is a set of colors optimized for use on display devices with a limited number of colors. A palette is defined by a palette resource or a palette data structure, each of which describes for each color in the palette the RGB color value, how the color is to be used, and its tolerance value if the color only needs to be approximated. See “The Palette Resource” in the chapter “Palette Manager Reference” in *Advanced Color Imaging Reference* on the enclosed CD for information on the structure of a palette resource. See “The Palette Structure” in the chapter “Palette Manager Reference” in *Advanced Color Imaging Reference* on the enclosed CD for details on the palette structure; this structure contains a series of color information structures that contain the actual information on each color, so you may want to look at “The Color Information Structure” as well.

The Palette Paradigm

Palettes are easy to create, either as resources or as data structures within your application. Once you create a palette and attach it to a particular window, the Palette Manager manages the window’s colors. Whenever your window is activated, the Palette Manager checks each graphics device that your window overlaps and ensures that your colors are available. In allocating colors for your window, the Palette Manager takes into account the importance you place on the various colors in the palette as well as the capabilities of the display devices your window touches.

There are several ways to create a palette. You can do any of the following:

- Place a palette resource (a resource of type ‘`pltt`’) in the resource fork of your application and create a palette from the color record information in the resource.
- Copy a palette from another palette.
- Build a palette from a color table.
- Create a palette by either of the methods just mentioned and then use the `SetEntry` and `SetUsage` functions to modify the colors in the palette and how they are used.

If you create a palette resource for a window in your application, and you assign the same ID to the resource and to the window, the Window Manager automatically calls the Palette Manager to attach the palette to the window when the window is first opened. If you create a palette in code (or from a

Palette Manager

resource that does not have the same ID as a window), you use the `SetPalette` or `NSetPalette` function to attach the palette to a particular window. If you wish, you can create a default palette to use for all windows in your application. You can create a default resource (a resource of type 'pltt' with ID = 0) from which to create the default palette or you can use the `SetPalette` or `NSetPalette` function to assign a default palette. See “Designating a Default Palette for Your Application” on page 1-29 for information on how to do this.

In the palette resource, or in the palette that you create, you specify the RGB colors your application needs. You can indicate whether each color must be matched exactly and, if not, how close a match is required. You can tailor your palettes to different possible video devices—indicating, for example, that certain colors in the palette should be used with 4-bit pixel depths, that a different set should be used with 8-bit pixel depths, and that neither set should be used with grayscale devices.

The Palette Manager can handle different screen depths across multiple devices. If the user moves your application window so that it overlaps one grayscale screen, one indexed-color screen, and one direct-color screen, the Palette Manager chooses appropriate grays and colors for all three.

The Palette Manager has access to all palettes used by all windows throughout the system. A set of default color tables for devices of various depths ensures that the Palette Manager always returns to a known set of colors when an application terminates. When your application begins executing, it executes in an environment equipped with as broad a range of colors or grays as the hardware allows; that is, a broad range of colors is always provided but the number of actual colors is limited by the pixel depth of the screen. For example, the default color table for a 4-bit pixel depth screen provides a range of 16 colors, the default color table for a 16-bit pixel depth screen provides a range of 256 colors, and so on.

Colors in a Palette

When the user activates a window, the Palette Manager examines the window and its palette to determine how many screens the window touches and whether colors need to be loaded into any device’s color lookup table. If your window requires 180 shades of green, for example, chances are the current device color tables lack the necessary colors. Whether the Palette Manager must change a color table depends on what colors are in it already, what colors you ask for, and the categories into which your colors fall.

When using Color QuickDraw you specify colors as RGB values. An RGB color is defined by its red, green, and blue components, which you specify in an `RGBColor` data structure. The brightest white consists of three maximum integer values (65535); black is three minimum values (0); grays are any three equal values between white and black. See the “Color QuickDraw” chapter of *Inside Macintosh: Imaging With QuickDraw* for further information about RGB colors and the `RGBColor` structures you use to specify them.

You may know the RGB values of the colors you need, or you may determine them by trial and error or by using a color picker.

Generally, the Palette Manager comes into play implicitly in your application; that is, you first create a palette and attach it to a window. Then, when you display an image or draw in the window, if any of the colors you specify is not available in the color table, the Palette Manager loads them for you, if it can.

The Palette Manager also provides two functions, `PMForeColor` and `PMBBackColor`, that enable you to draw explicitly with a palette’s colors. Each of these functions takes a palette entry as an argument. You can use `PMForeColor` and `PMBBackColor` with any type of color (see the rest of this section for a description of the different usage categories that determine the type of a palette color, including courteous, tolerant, animated and explicit). You must use `PMForeColor` and `PMBBackColor` when drawing with animated colors. See “Drawing With a Palette’s Colors,” beginning on page 1-31 for more information on when and how to use the Palette Manager drawing functions. For a complete description of the `PMForeColor` and `PMBBackColor` functions, see the chapter “Palette Manager Reference” in *Advanced Color Imaging Reference* on the enclosed CD.

The Palette Manager tracks colors in usage categories, which you specify to control the way the Palette Manager allocates your palette’s colors. When you create your palette, you assign usage categories to colors with the usage constants in the color information record—see “The Color Information Structure” in the chapter “Palette Manager Reference” in *Advanced Color Imaging Reference* on the enclosed CD. You can change the categories by using Palette Manager functions. You can assign any of the following categories to each color in your palette. The four categories, courteous, tolerant, animated,

and explicit define the types of colors that the Palette Manager provides. (Note that some of the categories can be combined.)

- A color specified as **courteous** accepts whatever value the Color Manager determines to be the closest match currently available in the device color table. On indexed devices, the Palette Manager lets the Color Manager select appropriate pixel values from those already in the device color table. On direct devices, the Palette Manager matches courteous colors as closely as the hardware allows. Courteous colors have no special properties, but their use offers you a convenient way to name and hold color collections.
- A color specified as **tolerant** also accepts the Color Manager's choices on an indexed device, but unlike a courteous color, a tolerant color specifies an acceptable range for color matching. If no color in the device's color table falls within that range, the Palette Manager loads the color required. If you specify a tolerance of 0, the color must match exactly. On direct devices, the Palette Manager matches tolerant colors as closely as the hardware allows.
- A color specified as **animated** is used for special color animation effects, as described in "Animated Colors," beginning on page 1-12. Animated colors are reserved by a palette until its window is closed, and until then their spaces are unavailable to (and can't be used to match) any other request for color. The effects of color animation depend on the existence of a device color table, and, because a direct device doesn't have a color table, color animation has no effect on a direct device's display. If your window spans two devices, one indexed and one direct, the Palette Manager is dexterous enough to animate the portion on the indexed device's screen.
- An **explicit** color specifies an index value rather than an RGB color and always generates the corresponding entry from the device's color table. Explicit colors are useful if you wish to display the contents of a color table—for example, to display to a user some or all of the colors actually available. If you specify an index value greater than those available in the palette, the Palette Manager wraps the entry and selects an index value that is within range.

The next two groups of categories, `inhibited` and `pmWhite/pmBlack` are always used with one or more of the four color-type usage categories just defined.

- A color specified as **inhibited** is prevented from appearing on color and grayscale devices of specified pixel depths. You always use inhibited colors in combination with other usage categories. You can create a large palette—for example, with two different sets of color ranges, one optimized for a 4-bit device, the other optimized for an 8-bit device—and then inhibit colors on the devices for which they are not intended.
- Colors specified as `pmWhite` or `pmBlack` are assigned to white or black, respectively, on 1-bit devices. If your application is working with red and dark blue, for example, both might get mapped to black on a 1-bit device. By assigning `pmWhite` to one and `pmBlack` to the other, you assure that they are always distinct. You can combine these categories with other usage categories, but combining them with each other is undefined.

You can combine several color usage categories. You can specify that a color is both tolerant and explicit, for example, which means that your RGB color, or a tolerably close match, is placed in the color table at the index corresponding to that palette entry (as opposed to merely being available somewhere in the table) on all devices that touch the window. See “Combining Color Usage for an Entry,” beginning on page 1-16 for additional information on combining two or more color usage categories.

Typically, you create a palette in which all colors have the same usage, and then if any entries need to be in a different category, you change them with Palette Manager functions.

The following sections describe the usage categories in more detail.

Courteous Colors

Courteous colors have no special properties, but they can serve as convenient placeholders. If your application uses a small number of colors, you can order them in a palette according to your preference and designate them as courteous.

Colors with specified usage categories that can’t be satisfied by the Palette Manager default to courteous colors. This occurs, for example, when drawing to a direct device or one with a fixed device color table.

Suppose you have an open window named `myColorWindow` that has a palette resource consisting of a set of eight colors: white, black, red, orange, yellow,

Palette Manager

green, blue, and violet, in that order, each with its color usage specified as courteous, as shown in Table 1-1.

Table 1-1 A courteous palette

Index	RGB value	Usage
0	White	courteous
1	Black	courteous
2	Red	courteous
3	Orange	courteous
4	Yellow	courteous
5	Green	courteous
6	Blue	courteous
7	Violet	courteous

The following example paints the rectangle `myRect` in yellow (palette entry 4, where white is 0).

```
SetPort (myColorWindow);
SetPalette (myColorWindow, srcPalette, TRUE);
PmForeColor (4);
PaintRect (myRect);
```

This is exactly analogous (if the usage is courteous) to the following sequence of Color QuickDraw functions, where `yellowRGB` is of type `ColorSpec`:

```
yellowRGB.red = $FFFF;
yellowRGB.green = $FFFF;
yellowRGB.blue = $0000;

SetPort (myColorWindow);
SetPalette (myColorWindow, srcPalette, TRUE);
RGBForeColor (yellowRGB);
PaintRect (myRect);
```

Tolerant Colors

Tolerant colors allow you to change the current color environment if the available colors are not sufficiently close to those your application needs. When your window becomes the frontmost window on a device, its palette's colors are given preference. Each tolerant color is compared to the best match available in the current color environment. (In a multiscreen environment this comparison is done for each device on which the window is drawn.) When the difference between your color and the best available match is greater than the tolerance you specify, the Palette Manager loads an exact match into the device color table.

The Palette Manager compares the tolerance value associated with each palette entry to a measure of the difference between two RGB color values. This difference is an approximation of the distance between the two points as measured in a Cartesian coordinate system where the axes are the unsigned red, green, and blue values. The distance formula used is

$$\Delta RGB = \text{maximum of } (abs(\text{Red1} - \text{Red2}), abs(\text{Green1} - \text{Green2}), abs(\text{Blue1} - \text{Blue2}))$$

A tolerance value of \$0000 means that only an exact match is acceptable. (Any value of \$0xxx other than \$0000 is reserved and should not be used in applications.) A value of \$5000 is generally sufficient to allow matching without updates in well-balanced color environments, such as those provided by the default palettes.

The color needs of your application determine the tolerance value to set for palette entries. For example, if your application is a drawing program in which exact colors are required, then you will specify a tolerance of zero for your palette entries. On the other hand, if exact matching is not a requirement, specifying a higher tolerance such as \$5000 is a good idea because it eliminates updates which can cause annoying screen flashes and also alter the color environment for other applications.

If your palette requires more colors than the number of unreserved table indexes, the Palette Manager checks to see if some other palette has reserved indexes for animation. If so, it cancels their reservation and makes their indexes available for your palette.

If you ask for more colors than are available on a device, the Palette Manager cannot honor your request. Color requests that can't be met default to courteous colors, and the Color Manager selects the best color available. That selection will, of necessity, match one of the colors elsewhere in your palette, because the Palette Manager runs out of colors only after it has given your

palette all that are available. This function works as well as possible for a given device, but of course, works better if your window is moved to a device of greater pixel depth where the request can be met.

Note that two tolerant entries may match to the same index even if space isn't the problem. For instance, when all indexes are initially assigned to black, activating a palette with 256 shades of gray with tolerance of \$2000 uses up four indexes, that being sufficient to match all 256 shades within a tolerance of \$2000. If the tolerance value were decreased to \$1000, then eight indexes would be altered.

On direct devices, tolerant entries always match as close to exactly as the hardware allows (to the first 8 bits of each component).

Animated Colors

Animated colors allow you to create color-table animation effects while lessening the disturbance caused other windows.

One way to change the color of an object on the screen is to change the pixel values in the object's part of the pixel map—you draw it again in a different color. In certain situations, you can get the same effect at less cost in processing and memory by changing the colors in the video device's color table instead. All pixel values corresponding to the altered indexes immediately appear on the display device in a new color. By careful selection of index values and the corresponding colors, you can achieve a number of special animation effects.

Note that no objects move in this animation; rather, the animation gives the appearance of motion, like the lights of a movie marquee.

To use an animated color, you must first draw with it using the `PmForeColor` or `PmBackColor` function. To create color-table animation, you then change that entry's RGB color by using the `AnimateEntry` function. You can animate a contiguous set of colors by using the `AnimatePalette` function to supply RGB colors from a color table.

The way the Palette Manager reserves indexes for animated colors creates some side effects. The Palette Manager first checks each animated color to see if it already has a reserved index for the target device. If it does not, the Palette Manager checks all windows and reserves the least frequently used indexes for your palette. (This reservation process is analogous to that used by the Color Manager function `ReserveEntry`.) The device's index and its corresponding color value are removed from the matching scheme used by Color QuickDraw; you cannot draw with the color by calling `RGBForeColor`. (However, when you

call `PmForeColor`, the Palette Manager locates the reserved index and configures your window's port to draw with it.) On a multiscreen system the index reserved is likely to be different for each device, but this process is invisible to your application.

After reserving one device index per device for each animated color it detects, the Palette Manager changes the color environment to match the RGB values specified in the palette.

The Palette Manager returns the indexes used by your animated entries to each screen device when any of the following occur:

- A window owning those animated entries moves off of that screen.
- Your application changes the usage of an animated color.
- Your application disposes of the palette owning those entries (merely hiding a window does not release its entries).
- Your application quits.

The Palette Manager replaces previously animated indexes with the corresponding colors from the default color table for that device.

The Palette Manager receives notice when the screen depth changes, so that it can take appropriate action at that time, such as setting color tables to their defaults.

Displaying Animated Colors on Direct Devices

Color-table animation doesn't work on a direct device because it has no color table. To present the best appearance, for example, on a window that spans an indexed device and a direct device, the Palette Manager records two colors in the `ciRGB` field of the `ColorInfo` structure: the last color the entry was set to by the `SetEntryColor` function and the last color the entry was set to by the `AnimateEntry` or `AnimatePalette` function. In the `ColorInfo` structure, the high bytes of the components in the `ciRGB` field reflect the animated color, and the low bytes contain the color set by `SetEntryColor`. (The `GetEntryColor` function returns the last color to which the entry was animated.) When you draw with an animated color on a direct device (or on any device on which the animated color was not allocated and reserved), then the color set by `SetEntryColor` is used. This allows successive updates of an animated image on a direct device to match correctly. A side effect is that `GetEntryColor` does not necessarily return an exact match of the color originally set (only the top 8 bits are an exact match).

▲ WARNING

This internal usage of color information fields may change. For maximum safety, use the functions `SetEntryColor`, `SetEntryUsage`, `GetEntryColor`, and `GetEntryUsage`. ▲

Explicit Colors

Use explicit colors when your primary concern is the index value rather than the color stored at that index.

Explicit colors cause no change in the color environment. For indexed devices, the Palette Manager ignores the RGB value in a palette if a color is an explicit color. When you draw with an explicit color, you get the color that is currently at the entry in the device color table whose index corresponds to the explicit color's position in the palette. When you call `PmForeColor` with a parameter of 12, it places a value of 12 into the foreground color field of your window's color graphics port. (Since the value wraps around the table, the value placed into the foreground field would be

12 modulo ($\text{maxIndex} + 1$)

where *maxIndex* is the maximum available index for each device under consideration.)

On direct devices an explicit entry produces the color for that entry in the palette.

You can use explicit colors to monitor the color environment on an indexed screen device. For example, you could draw a 16-by-16 grid of 256 explicit colors in a small window. Whatever colors appear are exactly those in the device's color table. If color-table animation is taking place simultaneously, the corresponding colors in the small window animate as well. If you display such a window on a 4-bit device, the first 16 colors match the 16 colors available in the device, and each row thereafter is a copy of the first row.

Inhibited Colors

The Palette Manager recognizes six inhibited usage categories that give you control of which palette entries can and cannot appear on depths of 2, 4, and 8 bits per pixel, on color or grayscale devices. The categories are specified using these constants:

```
enum {
    pmInhibitG2      = $0100;    /* inhibit on 2-bit grayscale device
    */
    pmInhibitC2      = $0200;    /* inhibit on 2-bit color device */
    pmInhibitG4      = $0400;    /* inhibit on 4-bit grayscale device
    */
    pmInhibitC4      = $0800;    /* inhibit on 4-bit color device */
    pmInhibitG8      = $1000;    /* inhibit on 8-bit grayscale device
    */
    pmInhibitC8      = $2000;    /* inhibit on 8-bit color device */
};
```

Here is an example of how these categories can be combined:

```
myColor8Usage = SetEntryUsage(myPalHandle, 270, pmAnimated+
    pmExplicit+pmInhibitG2+pmInhibitC2+pmInhibitG4+pmInhibitC4+
    pmInhibitG8, 0);
```

This example sets the usage of entry 270 of the palette specified by `myPalHandle` to the combined usages of animated and explicit, to be allocated only on color 8-bit devices. See “Selecting the Right Color Set,” beginning on page 1-25 for a further example of inhibiting particular colors on different types of devices.

You should always inhibit tolerant colors on grayscale devices. The default color table on a grayscale device is an evenly spaced gray ramp from black to white. Since this is usually the best possible spread on a grayscale device, you could specify all three inhibited grayscale categories.

As another example, on a 4-bit device you might want to allocate 14 tolerant colors, while on an 8-bit device there are sufficient indexes that you can also use a number of animated colors. By inhibiting the animated entries on 4-bit devices, you ensure that your 14 tolerant colors are allocated. Merely sequencing the palette doesn’t solve this problem because the animated colors always take precedence over the tolerant colors.

Combining Color Usage for an Entry

You must always combine the inhibited usage category with some other usage category. In addition, you can combine the explicit usage category with the tolerant and animated categories if you wish.

The main purpose for using explicit colors is to provide a convenient interface to color table indexes. The `PmForeColor` function configures the color graphics port to draw with the index of your choice. So that you can easily create effective explicit palettes, two color usage categories can be combined: `pmTolerant + pmExplicit` and `pmAnimated + pmExplicit`.

The `pmTolerant + pmExplicit` combined usage means that you get the color you want at the index you want, across all devices that the window touches. As with `pmTolerant`, other windows may use those colors in their displays.

The `pmAnimated + pmExplicit` combined usage means that you get the color you want at the index you want, across all devices that intersect the window, but windows that don't share the palette can't use that index. The entry can be animated by a call to the `AnimateEntry` function.

Since the value of an explicit entry is treated as the entry modulo the bit depth, index collisions can occur between entries of the same usage within a palette. In this case, the lower-numbered entry gets the index. For example, if palette entries 1 and 17 were both `pmAnimated + pmExplicit`, then on a 4-bit screen, entry 1 would get index 1, and entry 17, although it wraps around to 1, would get nothing.

Unallocated `pmTolerant + pmExplicit` colors revert to `pmTolerant`. Unallocated `pmAnimated + pmExplicit` colors revert to `pmCourteous`.

Sequencing the Entries

Using the inhibited usage categories is the best way to be sure that the right colors are available for screens of different depths, but in many situations you can achieve the same effect with a single set of colors if you sequence the colors in the palette or arrange them according to the screen depth of the device that uses them, from least to greatest depth.

Color QuickDraw, to support standard QuickDraw features, puts white and black at the beginning and end, respectively, of each device's color table, and the Palette Manager never changes them. Thus the maximum number of indexes available for animated or tolerant colors is really the maximum number of indexes minus 2.

After white and black, you should assign the next two colors to the two you wish to have if the device is a 2-bit device. Likewise, the first 16 colors should be the optimal palette entries for a 4-bit device, and the first 256 colors should be the optimal palette entries for an 8-bit device. You should inhibit colors for grayscale devices.

How the Palette Manager Allocates Colors for Display

The colors available when your application is running depend on the pixel depths and color capabilities that the attached cards and screens can support, as well as what those devices are actually set to by the system or by the user.

Since your palette may define more colors than the hardware can display, the Palette Manager allocates device color-table entries for your requests according to the priority of their usage. Prioritizing occurs when the `ActivatePalette` function is called, which occurs automatically when your window becomes the frontmost window. (You may also call `ActivatePalette` yourself after changing one or more of the palette's colors or usage categories.)

The Palette Manager first allocates animated colors that are also specified as explicit. Colors that are specified as both tolerant and explicit are allocated next.

The Palette Manager allocates animated colors next. Starting with the first entry in your window's palette (entry 0), the Palette Manager checks to see if it is an animated entry. The Palette Manager checks each animated entry to see that the entry has a reserved index for each appropriate device and selects and reserves an index if needed. This process continues until all animated color requests have been satisfied or until the available indexes are exhausted.

The Palette Manager handles tolerant colors next. It assigns each tolerant color an index until all tolerant color requests have been satisfied. The Palette Manager then calculates for each entry the difference between the desired color and the color associated with the selected index. If the difference exceeds the tolerance you have specified, the Palette Manager marks the selected device entry to be changed to the desired color.

Since explicit colors designate index values, not the colors at those index locations, and since courteous colors are amenable to being assigned any RGB value, neither is considered during prioritizing (except for explicit combined with animated).

When the Palette Manager has matched as many animated and tolerant entries as possible, it checks to see if the current device color table is adequate. If

Palette Manager

modifications are needed, the Palette Manager overrides any calls made to the Color Manager outside the Palette Manager and then calls the Color Manager to change the device's color environment accordingly (with the `SetEntries` function).

Finally, if the color environment on a given device has changed, the Palette Manager checks to see if this change has affected any other window in the system. If another window is affected, the Palette Manager checks that window to see if it specifies an update in the case of such changes. Applications can use the `SetPalette`, `NSetPalette`, or `SetPaletteUpdates` function to specify whether a window should be updated when its environment has been changed because of actions by another window. (If so, the `InvalidRect` function, described in the "Window Manager" chapter of *Inside Macintosh: Macintosh Toolbox Essentials*, updates the window using the boundary rectangle of the device that has been changed.)

How the Palette Manager Restores the Color Environment

When a window closes, the Palette Manager resets each display device to the default color table for that depth, except for those indexes still reserved by another application. (Eventually, the application that owns those indexes will terminate or voluntarily release the indexes.) You can run a long sequence of wildly animated color-stealing programs, quit them all, return to the Finder, and find every screen fully restocked with default system color tables. (But if an application calls the Color Manager function `ProtectEntry` to lock a device index, the Palette Manager cannot restore the default color tables.)

The Palette Manager provides default color tables for differing screen devices:

Screen device

Default color table

Any device in gray mode or 1 bit deep

A grayscale ramp; that is, an evenly spaced range from white in index 0 to black in the last index (only white and black in 1-bit mode).

A color device in 2-bit mode

Indexes 0 to 3 contain white, 50 percent gray, the highlight color, and black, respectively.

A color device in 4-bit mode

The 'clut' system resource with a resource ID of 4. It could be in the system file or stored in ROM. If the color closest to the highlight color differs from it by more than \$3000 in any component, the color is averaged with the highlight color.

A color device in 8-bit mode

The 'clut' system resource with a resource ID of 8. It could be in the system file or stored in ROM.

The 'clut' resource IDs 1, 2, 4, and 8 are the standard color lookup tables for those bit depths; they are shown in Plate 7, “The colors of the default color tables” in the front of this book.

The 'clut' resource IDs 34, 36, and 40—the bit depth plus 32—are grayscale ramps for bit depths of 2, 4, and 8.

The default color lookup tables with the highlight color added are 'clut' resource IDs 66, 68, and 72—that is, the bit depth plus 64. To get these color lookup tables, use the `GetCTable` function (not `GetResource`), as described in the chapter “Color QuickDraw” of *Inside Macintosh: Imaging With QuickDraw*.

Using the Palette Manager

To use the Palette Manager you generally do one of the following:

- Create a palette in any of several ways.
- Modify the colors and usages of the colors in a palette.
- Assign a palette to a window or assign a default palette to multiple windows.

After creating a palette and assigning it to a window, you can do the following:

- Draw with the palette's colors.
- Animate the colors in a palette.
- Draw or animate in an offscreen graphics world.

These tasks are explained in the rest of this chapter. This chapter also describes how and when to dispose of palettes and how to restore the default color table for an application. However, Palette Manager functions are dependent on the availability of Color QuickDraw. Therefore, before calling any of the functions described in the following sections, your application should 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. Listed here are the various constants and the versions of QuickDraw that they represent.

```
/* quickdraw version */
gestaltOriginalQD = 0x000,      /* original 1-bit QD */
gestalt8BitQD = 0x100,          /* 8-bit color QD */
gestalt32BitQD = 0x200,         /* 32-bit color QD */
gestalt32BitQD11 = 0x210,       /* 32-bit color QDv1.1 */
gestalt32BitQD12 = 0x220,       /* 32-bit color QDv1.2 */
gestalt32BitQD13 = 0x230,       /* System 7: 32-bit color QDv1.3 */
```

Your application can also use the `Gestalt` function with the selector `gestaltSystemVersion` to check for the system software version under which your application is running. The functions `SaveFore`, `RestoreFore`, `SaveBack`,

`RestoreBack`, `ResizePalette`, and `RestoreDeviceClut` require System Software version 6.0.5 or greater:

Creating Palettes

This section shows you several different ways to create a palette. The first example uses the `NewPalette` function in a loop to create a 16-color palette with 14 shades of red plus black and white. The second example shows how to create a resource file containing the same 16 colors and shows you how to select the right color set by inhibiting certain colors on different screens. It also shows how to copy a palette from a color table and assign the palette to a window.

Creating a Palette in Code

The Palette Manager is particularly useful when your application needs to display an image that uses a narrowed or skewed color set. For example, suppose that your application displays PICT images on a 4-bit depth screen and that a particular image contains only shades of red. By default, the system provides a range of colors to use. If your application uses these 16 default colors to display the red PICT file, the display will lose all subtlety. With the Palette Manager, you can create a special palette containing red hues only to display this PICT image.

Note

This example assumes that the PICT image to display uses shades of red. One way to obtain information about a picture to display is by using the Picture Utilities, described in the chapter “Pictures” of *Inside Macintosh: Imaging With QuickDraw*. The Picture Utilities allow you to obtain various information about a picture to display, including the most used colors in the picture. The Picture Utilities are available in system software version 7.0 and later. ♦

Listing 1-1 shows how to create a palette containing shades of red only.

Listing 1-1 Creating a red palette

```

PaletteHandle DoMake14RedPalette (void)
{
    long                iterator;
    PaletteHandle       myPalette;
    RGBColor            color;

    myPalette = NewPalette(14, nil, pmTolerant, 4000);
    /* check here for nil result */

    color.green = 0;
    color.blue  = 0;
    for (iterator = 0; iterator < 14; iterator++)
    {
        /* range red component from 1/14 to 14/14 */
        /* iterator is a long, so it can safely be multiplied by 65535 */
        color.red = (iterator+1) * 65535 / 14;
        SetEntryColor(myPalette, iterator, &color);
    };
    return ( myPalette );
}

```

In Listing 1-1, the `DoMake14RedPalette` function creates a new 14-color palette containing 14 shades of red. The Palette Manager always guarantees that black and white are available to an application so that menus, windows, and other such things display properly. Therefore, you can load at most 14 new colors from a palette into a 16-color lookup table because the Palette Manager will not overwrite white (the first entry) or black (the last entry).

After defining three variables, the `DoMake14RedPalette` function calls the `NewPalette` function to create a palette large enough to hold 14 colors. The `nil` value passed as the second parameter specifies that the palette not be created from a color lookup table. In this case, the Palette Manager sets all three fields of each color to 0 (black). In essence, `NewPalette` has created a palette template. The `SetEntryColor` function later fills in, or changes, the color value for each color in the palette. The `NewPalette` function defines each color as tolerant with a tolerance of \$4000.

Because this palette contains shades of red only, the green and blue fields of the `color` variable are set to 0. The code then defines 14 shades of red in a loop by dividing the maximum red value (65535) by 14 and adding 1/14 of 65,535 each

time through the loop. Each time through the loop, the `SetEntryColor` function places a shade of red at the next palette entry (identified by `iterator`, the iteration variable). When the loop is finished, the first 14 entries in the palette created by the `NewPalette` function contain 14 evenly distributed shades of red.

Note

When the Palette Manager uses this palette—after you have attached it to a window with the `SetPalette` or `NSetPalette` function—it looks at the colors of the color lookup table for the 4-bit display device and determines which of the 14 colors in the palette it must load. It then loads as many of the 14 palette entries as necessary to create the 14 shades of red defined in the palette. The Palette Manager keeps the first entry in the table as white and the last as black and uses the other 14 indexes as necessary to load the shades of red. See “How the Palette Manager Allocates Colors for Display” on page 1-17 for more information on how the Palette Manager loads colors from a palette into a color lookup table. ♦

Creating a Palette in a Resource File

The format of a palette resource (type `'pltt'`) is an image of the palette structure minus the private fields. The private fields in both the header and in each `ColorInfo` record are created by the `GetNewPalette` function and are reserved for future use.

Listing 1-2 shows a palette resource with 16 entries as it would appear within a resource file. The black and white entries each have a tolerance value of 0, meaning that their color should be matched exactly. The shades of red have a tolerance of \$4000.

Listing 1-2 A palette ('pltt') resource

```
resource 'pltt' (128, "Simple Palette") {
{   /* array ColorInfo: 16 elements */
/* [1] white */
65535, 65535, 65535, pmTolerant, 0,
/* [2] 2 through 15 are shades of red */
4681, 0, 0, pmTolerant, $4000,
```

Palette Manager

```

/* [3] */
9362, 0, 0,          pmTolerant, $4000,
/* [4] */
14043, 0, 0,         pmTolerant, $4000,
/* [5] */
18724, 0, 0,         pmTolerant, $4000,
/* [6] */
23405, 0, 0,         pmTolerant, $4000
/* [7] */
28086, 0, 0,         pmTolerant, $4000,
/* [8] */
32768, 0, 0,         pmTolerant, $4000,
/* [9] */
37449, 0, 0,         pmTolerant, $4000,
/* [10] */
42130, 0, 0,         pmTolerant, $4000,
/* [11] */
46811, 0, 0,         pmTolerant, $4000,
/* [12] */
51492, 0, 0,         pmTolerant, $4000,
/* [13] */
56173, 0, 0,         pmTolerant, $4000,
/* [14] */
60854, 0, 0,         pmTolerant, $4000,
/* [15] */
65535, 0, 0,         pmTolerant, $4000,
/* [16] black */
0, 0, 0,            pmTolerant, 0
    }
};

```

Use the `GetNewPalette` function to obtain a 'pltt' resource; it initializes private fields in the palette structure. (Don't use the Resource Manager function `GetResource`.)

The palette defined by the palette resource in Listing 1-2 is identical to the palette created in code in Listing 1-1 in the previous section except that white and black are explicitly placed in the palette in Listing 1-2. This palette contains black and white and 14 shades of red. The shades of red are defined by setting the green and blue values of each color entry to 0 and distributing the red color

in 14 increments from the lowest to the highest value (65535). One possible use of this palette is to display a PICT file that contains only shades of red.

Selecting the Right Color Set

Different types of screens (for example, screens with different bit depths) often require different color sets to best display the same image. This section, like the previous section, uses an example of creating a palette to display an image in various shades of red. In addition, the code has been modified to display different colors on different screens.

Listing 1-3 shows the code to display 14 shades of red on a 4-bit color screen, 254 shades of red on an 8-bit color screen, and no color requests at all on 2-bit screens or grayscale screens.

Listing 1-3 Displaying different colors on different types of screens

```
PaletteHandle MyMakeRedPalette (void)
{
    long                iterator;
    PaletteHandle       myPalette;
    RGBColor            color;

    myPalette = NewPalette(254+14, nil, 0, 0);
    /* check here for nil result */

    color.green = 0;
    color.blue = 0;
    /* make 14 reds that are inhibited on all
       screens except 4-bit color */
    for (iterator = 0; iterator < 14; iterator++)
    {
        /* range red component from 1/14 to 14/14 */
        /* iterator is a long, so it can safely be multiplied by 65535 */
        color.red = (iterator+1) * 65535 / 14;
        SetEntryColor(myPalette, iterator, &color);
        SetEntryUsage(myPalette, iterator, pmTolerant+pmInhibitC2+
                               pmInhibitG2+pmInhibitG4+
                               pmInhibitC8+pmInhibitG8, 0);
    };
};
```

Palette Manager

```

/* make 254 reds that are inhibited on all
   screens except 8-bit color */
for (iterator = 0; iterator < 255; iterator++)
{
    /* range red component from 1/254 to 254/254 */
    /* iterator is a longint, so can safely be multiplied by 65535 */
    color.red = (iterator+1) * 65535 / 254;
    SetEntryColor(myPalette, 14+iterator, &color);
    SetEntryUsage(myPalette, 14+iterator, pmTolerant+pmInhibitC2+
                                                         pmInhibitG2+pmInhibitG4+
                                                         pmInhibitC4+pmInhibitG8, 0);
};
return ( myPalette );
}

```

In Listing 1-3, the `MyMakeRedPalette` function creates a palette that contains shades of red in order to optimally display an image that contains shades of red only. The palette contains two less colors than the maximum (14 for a 4-bit screen and 254 for an 8-bit screen) so that black and white are available in the color table.

After defining three variables, the `MyMakeRedPalette` function calls the `NewPalette` function to create a palette large enough to hold 268 colors (254 plus 14). The `nil` value passed as the second parameter specifies that no color table be used to create the template, in which case all the colors in the palette are set to black. The two `for` loops that follow use the `SetEntryColor` and `SetEntryUsage` functions to change the colors and their use in the palette.

The first loop creates 14 shades of red. The green and blue values for all the colors are already set to 0. The `SetEntryColor` function is called in a loop to set the 14 red values. It specifies 1/14 increments up to the maximum value of 65535. The `SetEntryUsage` function specifies that each color is tolerant (it uses the `pmTolerant` constant to do this) and that the tolerance is 0 so that each color must match exactly.

The `SetEntryUsage` function also specifies that these 14 colors are to be used only on a 4-bit screen. It does this by inhibiting these colors on all other indexed screens using the constants `pmInhibitC2` (inhibit on a 2-bit color screen), `pmInhibitG2` (inhibit on a 2-bit grayscale screen), and so on for the other types of indexed screens.

The second loop creates 254 shades of red. It uses the `SetEntryColor` function as the first loop does with the only difference being that the red color value is

incremented 1/254 at each iteration rather than 1/14. As in the first loop, the `SetEntryUsage` function specifies that each color is tolerant with a tolerance of 0 so that the colors must match exactly. The `SetEntryUsage` function also specifies that the colors are to be used only on an 8-bit color screen. It does this by inhibiting the colors on all other indexed screens.

Creating a Palette by Copying and Assigning It to a Window

You can create a palette from the colors in a color table by using the `NewPalette` function. It copies the colors from the table and assigns the usage and tolerance values you specify as function parameters. You assign a usage value and a tolerance value that apply to all of the palette's entries, but you can change individual entries with subsequent calls to the `SetEntryUsage` function, which can change both the usage and tolerance values of an entry. (If you call `NewPalette` without supplying a color table as a source, it creates a palette with all entries equal to black. You can then modify the entries as needed, using the `SetEntryColor` and `SetEntryUsage` functions.)

Note

Color tables define the colors that are available for pixel images on indexed devices. You can create color tables from either `ColorTable` data structures or color table ('clut') resources. See the chapter "Color QuickDraw" of *Inside Macintosh: Imaging With QuickDraw* for more information on color tables. ♦

For example, suppose you are drawing to an indexed device with a color table containing 14 colors plus black and white and you want a palette in which the fifth color must be matched exactly but the others need only be close. Listing 1-4 shows how to create the palette from the color table and then modify the tolerance value for the fifth entry (remember that color table and palette entries are numbered starting at 0).

Listing 1-4 Copying a color table to a palette

```

#define      clutID      68

void DoCreatePalette()
{
    CTabHandle      myColorTable;
    PaletteHandle    myPalette;
    WindowPtr       myWindow;

    myColorTable = GetCTable (clutID);

    /* create a new window */
    myWindow = NewCWindow(nil, &BaseRect, "\pUsing Palette
                                   Manager", TRUE, documentProc,
                                   (WindowPtr) -1, TRUE, nil);

    /* create a 16-color palette */
    myPalette = NewPalette(16, myColorTable, pmTolerant, $2000);
    /* modify the 5th entry */
    SetEntryUsage(myPalette, 4, 0);

    /* assign the palette to the window */
    SetPalette ((WindowPtr) myWindow, myPalette, TRUE);
}

```

After defining three variables (to hold the color table ID, the palette ID, and the window ID), the `DoCreatePalette` function uses the `GetCTable` function (described in the chapter “Color QuickDraw” of *Inside Macintosh: Imaging With QuickDraw*) to retrieve the default color-table ('clut') resource for a screen with a 4-bit pixel depth. (You can obtain the default color-table resource for a screen by adding 64 to the pixel depth of the screen. Therefore, 68 identifies the color-table resource for a 4-bit pixel depth screen.) The `NewCWindow` function (described in the chapter “Window Manager” of *Inside Macintosh: Macintosh Toolbox Essentials*) then creates a new color window.

The `NewPalette` function creates a palette from the color-table resource. All colors in the new palette have a tolerance of \$2000. The `SetEntryUsage` function changes the tolerance of the fifth entry such that an exact match is required.

Finally, the `SetPalette` function assigns the newly created palette to the newly created window.

Note

You can use either the `SetPalette` or `NSetPalette` function to assign a palette to a window. These functions are identical except that the `NSetPalette` function is more versatile in specifying whether the window is to receive updates when the color environment changes; it allows you to specify whether the window is to receive updates only when the window is active, updates only when it is not active, all updates, or none. The `SetPalette` function only allows the last two settings. ♦

If you want to change a window's palette momentarily and then restore the first palette, use the `GetPalette` function to obtain a handle to a window's current palette and then restore it with either `SetPalette` or `NSetPalette`.

Designating a Default Palette for Your Application

Your application can define a default palette for the Operating System to use for your windows. Defining a default palette is useful if all your windows use the same palette or if you use basic QuickDraw dialog and alert boxes.

There are two ways that you can define a default palette for your application: by creating a palette resource that the Palette Manager can then use to create the default palette or by using the `SetPalette` or `NSetPalette` function to assign a default palette.

You set a palette resource (a resource of type 'pltt') as your application's default by assigning it a resource ID of 0. The Palette Manager loads the application default palette and stores it in a low-memory global variable called `AppPalette`. When your application opens a new color window, the Palette Manager automatically looks for a palette resource from which to create a palette to assign to the window. It looks in the resource fork for a palette resource with the same ID as that of the window. If you haven't created a specific resource for the window and if you haven't assigned a palette to the window with the `SetPalette` or `NSetPalette` function, the Palette Manager creates a palette from the default application resource and assigns this palette to the window. If the Palette Manager cannot find the default application resource, it uses the system default palette resource ('pltt' ID = 0 in the System file). If the System file has no default palette, the Palette Manager creates a special two-entry palette (black and white) and assigns this palette to the window.

Palette Manager

IMPORTANT

If you are running an application from a development environment, such as MPW, you could have trouble automatically assigning the default application palette. The Palette Manager loads the default palette when a call is made to `InitPalettes` (generally, `InitWindows` makes this call). Then `InitPalettes` uses `Get1Resource` rather than `GetResource` to load the 'pltt' resource with ID equal to 0. The problem is that `Get1Resource` assumes your application is at the top of the resource chain and retrieves the resource only if it is in the first file. However, because the development environment, not your application, is at the top of the resource chain, the Palette Manager will be unable to load the default application resource. In this case, it loads the default system resource, if it exists, or the special two-entry palette instead. ▲

The other way to assign a default palette is to create a palette and then assign it with the `SetPalette` or `NSetPalette` function as follows:

```
defPalette = NewPalette (numcolor, mycolors, pmAnimated, 0x1500);
SetPalette ((WindowPtr) -1, defPalette, TRUE);
```

This method of assigning a default palette is useful if, for example, you give the user an opportunity in a menu or preference file to change the palette to another setting.

Once your application has set its color environment by calling `InitWindows` (or `InitPalettes` in unusual instances when there are no menus), you can find the default palette for your application by using the `GetPalette` function:

```
myPaletteHndl = GetPalette ((WindowPtr) -1);
```

Drawing With a Palette's Colors

You can use the `PmForeColor` and `PmBackColor` functions to specify foreground and background drawing using colors from your palette. The functions set the foreground and background colors of the current graphics port to the palette colors you specify so that subsequent drawing operations use them. (In effect, you substitute these functions for the Color QuickDraw functions `RGBForeColor` and `RGBBackColor`. Use the `RGBForeColor` and `RGBBackColor` functions to specify drawing with colors not contained in your palette.)

For courteous and tolerant entries, `PmForeColor` calls the `RGBForeColor` function using the RGB color of the palette entry. For animated colors, `PmForeColor` selects the recorded device index previously reserved for animation (if still present) and installs it in the color graphics port. The RGB foreground color field is set to the value from the palette entry. For explicit colors, `PmForeColor` places the value

`dstEntry modulo (maxIndex +1)`

into the color graphics port, where *maxIndex* is the largest index available in a device's color table. When multiple devices with different depths are present, the value of *maxIndex* varies appropriately for each device.

You can save and restore the current foreground and background colors by using the `SaveFore`, `RestoreFore`, `SaveBack`, and `RestoreBack` functions.

Animating a Window With a Palette

The Palette Manager provides functions that allow you to create color animation effects. Listing 1-5 shows a simple means of creating animation in your application.

Note

Color animation requires color tables. Therefore, you can create color animation effects on indexed screens only. Screens with 16-bit and 32-bit pixel depth do not have color tables and therefore cannot support Palette Manager animation. ♦

Listing 1-5 Animating with a palette

```

#define      clutID      150
#define      numcolor    256

void DoAnimate()
{
    CTabHandle      myColorTable, StoreCTab;
    PaletteHandle   myPalette;
    WindowPtr       myWindow;
    RGBColor        changeColor;

    myColorTable = GetCTable (clutID);

    /* create a new window */
    myWindow = NewCWindow(nil, &BaseRect,
                          "\pUsing Palette Manager",
                          TRUE, documentProc,
                          (WindowPtr) -1, TRUE, nil);

    /* create a 256-color palette */
    myPalette = NewPalette(numcolor, myColorTable, pmAnimate, 0);

    /* assign the palette to the window */
    SetPalette ((WindowPtr) myWindow, myPalette, TRUE);

    GetEntryColor (myPalette, 1, &changeColor);
    AnimatePalette (myWindow, StoreCTab, 2, 1, numcolor - 2);
    AnimateEntry (myWindow, numcolor - 1, &changeColor);
    PaletteToCTab (myPalette, StoreCTab);
}

```

In Listing 1-5, the `DoAnimate` function does some setup before performing the animation. It first declares variables for two color lookup tables, for the window and palette IDs, and for an RGB color. It retrieves a color table with the `GetCTable` function (described in the chapter “Color QuickDraw” of *Inside Macintosh: Imaging With QuickDraw*) and creates a new color window with the `NewCWindow` function (described in the chapter “Window Manager” of *Inside Macintosh: Macintosh Toolbox Essentials*). It then creates a palette from the color table using `NewPalette` and assigns this palette to the newly created window

using `SetPalette`. The palette contains 256 colors and the colors are all animated colors.

The first step in the animation is to use the `GetEntryColor` function to save the first color in the palette. Then `AnimatePalette` cycles through each color in the palette (except for black and white, which do not animate). The `AnimateEntry` function moves the saved color to the last entry in the palette. Finally, `PaletteToCTab` saves the new version of the palette to the color table for use during the next animation.

One thing you could do with this animation code is to put it in a loop. Because the first entry has been moved to the end, the animation, in effect, begins at the second entry during the second iteration, and so on for each time through the loop. The result is an animation that not only cycles through the palette but also displays a slight variation at each iteration by beginning with a different color.

Disposing of a Palette and Restoring the Color Table

To prevent memory leaks, it is good programming practice to use the `DisposePalette` function to dispose of palettes when your application is done with them. If you explicitly attach a palette to a window with the `SetPalette` or `NSetPalette` function, then your application owns the palette and must dispose of it when it is no longer needed. Note that a palette can be attached to more than one window, so closing a window doesn't necessarily mean that its palette is no longer needed.

If you call `GetPalette`, which makes a copy of a window's palette, you must also call `DisposePalette` when you no longer need the palette.

If a palette has been set up automatically by the Palette Manager and Window Manager because the palette resource has the same ID as the window (or if the palette was created from the default application palette resource), then these managers automatically dispose of the palette when the window goes away.

It is also good programming practice to use the `RestoreDeviceClut` function to restore the color table of all graphics devices to their default state whenever your application is switched out as the active application. You don't have to worry about switching to the Finder, because its colors are automatically restored upon switching from applications that use the Palette Manager; nor do you need to be concerned with applications that use the Palette Manager, because the Palette Manager provides the proper colors the moment a window in such an application comes to the front. However, if you switch to an

application that does not use the Palette Manager, its windows inherit your palette unless you have restored the default color tables for all the available display devices.

Using Palettes With Offscreen Graphics Worlds

You can attach a palette to an offscreen graphics world; however, the only Palette Manager color usage categories that you can use to specify the types of colors in such a case are `pmCourteous` and `pmBlack` and `pmWhite`. The other three color usage categories, `pmTolerant`, `pmAnimated`, and `pmExplicit`, are interpreted in an offscreen graphics world as `pmCourteous`, so they are of no use in this environment.

This section first explains why and how you might use `pmCourteous`, `pmBlack`, and `pmWhite` and then shows how to simulate the effects of the other usage modifiers that you cannot use with an offscreen graphics world.

After you've created a palette and assigned it to an offscreen graphics world, you can pass indexes from your palette to the `PmForeColor` and `PmBackColor` functions and then draw in the offscreen world. The advantage of using courteous colors for the offscreen graphics world is that otherwise you must hard-code colors into your code. If you or a software localizer wants to change the colors in the offscreen worlds, you can do so by changing the 'pltt' resource that defines the palette rather than making code changes and recompiling the application.

The `pmBlack` and `pmWhite` usage modifiers simply allow you to specify which colors map to white and which to black in a black-and-white environment. If your application is working with red and dark blue, for example, both might get mapped to black on a 1-bit device. By assigning `pmWhite` to one and `pmBlack` to the other, you assure that they are always distinct.

Explicit colors in a palette attached to an offscreen graphics world are interpreted as courteous. Therefore, instead of using a palette, you should convert your pixel value to an RGB color and use this as the foreground or background color. After setting the current graphics device to the offscreen graphics world to set the color environment, convert your pixel value to the corresponding RGB color using the `IndexToColor` function. Then you can draw by passing the RGB color to the `RGBForeColor` and `RGBBackColor` functions.

Both tolerant and animated colors are interpreted as courteous in an offscreen graphics world. To use these types of colors, you can change the color

environment of the offscreen graphics world by assigning it a whole new color table. Listing 1-6 shows how to do this.

Listing 1-6 Changing the color environment of an offscreen graphics world

```
#define      clutID      150
#define      numcolor    256

WindowPtr    myWindow;
CTabHandle   mycolors;
PaletteHandle srcPalette

GetGWorld (&SavePort, &SaveGD);
mycolors = GetCTable (clutID);
/* create a new window */
myWindow = NewCWindow(nil, &BaseRect, "", TRUE, zoomDocProc,
                      (WindowPtr) -1, TRUE, nil);
SetGWorld((CGrafPtr)myWindow, SaveGD);
DrawGrowIcon (myWindow);

(*mycolors)->ctFlags |= 0x4000;

/* create a 256-color palette */
srcPalette = NewPalette (numcolor, mycolors, pmCourteous, 0);
/* call DoSetInhibited to set color usage */
DoSetInhibited(pmCourteous);
SetPalette ((WindowPtr) myWindow, srcPalette, TRUE);

GetGWorld (&SavePort, &SaveGD);
err = NewGWorld (&offscreenGWorld, 8, &InitWindowSize,
                mycolors, nil, nil);

if (err)
    Debugger();
SetGWorld (offscreenGWorld, nil);
EraseRect (&InitWindowSize);
DrawPicture (ThePict, &InitWindowSize);
SetGWorld (SavePort, SaveGD);
```

Palette Manager

After defining constants and declaring variables, the code in Listing 1-6 calls the `QuickDraw GetGWorld` function to get and save the current graphics port so that you can restore it later. The `GetCTable` function retrieves a handle to a color table—from a 'clut' resource with ID 150—and stores it in the `mycolors` variable. You will use this color table for the offscreen graphics world and to create a palette.

The `NewCWindow` function creates a new window and the `SetGWorld` function assigns the current graphics port to this newly created window. The `DrawGrowIcon` function draws the window's size box.

The next piece of code:

```
(*mycolors)->ctFlags |= 0x4000;
```

sets bit 14 of the `ctFlags` field in the color table. Setting this bit synchronizes the color table to a palette. It does this by reinterpreting the `ctTable` field of the color table. Normally, this field contains an array of `ColorSpec` entries, each of which contains a *pixel* value and a color. After setting bit 14 of the `ctFlags` field, each array in the `ctTable` field contains an *index* value and a color.

The `NewPalette` function creates a new palette from the color table retrieved by the `GetCTable` function. It contains 256 colors. Initially, all the colors are courteous and must match exactly (have a zero tolerance value).

The `DoSetInhibited` function (whose code is not shown here) uses the Palette Manager `SetEntryUsage` function to change the usage categories of the palette when the usage of the palette changes. For example, it sets the colors in the palette to courteous, tolerant, explicit, or animated, depending on menu selections that a user makes. The `DoSetInhibited` function also inhibits certain colors in the palette depending on the bit-depth of the screen on which the window is currently displayed. In this way, the palette can contain entries for multiple screen bit-depths and will display properly on each type of screen. See “Selecting the Right Color Set,” beginning on page 1-25 for an example of how to create a palette that will display on screens with different bit-depths. The application calls `DoSetInhibited` at this point to set the palette for the appropriate screen depth. Since `pmCourteous` is passed to `DoSetInhibited`, the usage category is not being changed.

The `SetPalette` function assigns the newly created palette to the window created by `NewCWindow`. The `GetGWorld` function gets and saves the current graphics port so it can be restored later. The `NewGWorld` function creates an

offscreen graphics world and the `SetGWorld` function sets the current graphics port to the newly created offscreen graphics world.

The `EraseRect` and `DrawPicture` functions erase and draw in the offscreen graphics world.

The final `SetGWorld` function restores the graphics port to the window created by the `NewCWindow` function. (To move the offscreen drawing to the screen, use the `QuickDrawCopyBits` function—see the chapter, “QuickDraw Drawing,” in *Inside Macintosh: Imaging With QuickDraw* for an example of how to do this.) The colors from the offscreen world—which come from the color table—match the colors of the window—which are controlled by the palette—because the color table and the palette are identical and synchronized. The advantage of this synchronization is that your application doesn’t have to use the offscreen world’s color matching algorithm to synchronize the colors from the offscreen world with the colors of the palette. Having to do so would greatly slow down your application, and speed is one of the primary reasons to use an offscreen world for drawing in the first place.

Summary of the Palette Manager

Constants

```
enum {
    pmCourteous = 0,           /* Courteous color */
    pmTolerant = 0x0002,       /* Tolerant color */
    pmAnimated = 0x0004,       /* Animated color */
    pmExplicit = 0x0008,       /* Explicit color */
    pmWhite = 0x0010,          /* Use on 1-bit devices */
    pmBlack = 0x0020,          /* Use on 1-bit devices */
    pmInhibitG2 = 0x0100,
    pmInhibitC2 = 0x0200,
    pmInhibitG4 = 0x0400,
    pmInhibitC4 = 0x0800,
    pmInhibitG8 = 0x1000,
    pmInhibitC8 = 0x2000,

    /* NSetPalette Update Constants */
    pmNoUpdates = 0x8000,      /*no updates*/
    pmBkUpdates = 0xA000,      /*background updates only*/
    pmFgUpdates = 0xC000,      /*foreground updates only*/
    pmAllUpdates = 0xE000      /*all updates*/
};
```

Data Types

```
struct Palette {
    short pmEntries;           /*entries in pmTable*/
    short pmDataFields[7];     /*private fields*/
    ColorInfo pmInfo[1];
};

typedef struct Palette Palette;
typedef Palette *PalettePtr, **PaletteHandle;
```

```

struct ColorInfo {
    RGBColor ciRGB;           /*true RGB values*/
    short ciUsage;            /*color usage*/
    short ciTolerance;        /*tolerance value*/
    short ciDataFields[3];    /*private fields*/
};
typedef struct ColorInfo ColorInfo;

```

Functions

Initializing the Palette Manager

```

pascal void InitPalettes      (void);
pascal short PMgrVersion      (void);

```

Initializing and Allocating Palettes

```

pascal PaletteHandle NewPalette (short entries,CTabHandle srcColors,short
                                srcUsage,short srcTolerance);

pascal PaletteHandle GetNewPalette (
                                short PaletteID);

pascal void DisposePalette      (PaletteHandle srcPalette);

```

Interacting With the Window Manager

```

pascal void ActivatePalette     (WindowPtr srcWindow);
pascal PaletteHandle GetPalette (WindowPtr srcWindow);
pascal void SetPalette          (WindowPtr dstWindow,PaletteHandle srcPalette,Boolean
                                cUpdates);

pascal void NSetPalette         (WindowPtr dstWindow,PaletteHandle srcPalette,short
                                nCUpdates);

pascal void SetPaletteUpdates   (PaletteHandle p,short updates);
pascal short GetPaletteUpdates  (PaletteHandle p);

```

Drawing With Color Palettes

```

pascal void PmForeColor      (short dstEntry);
pascal void PmBackColor      (short dstEntry);
pascal void SaveFore         (ColorSpec *c);
pascal void RestoreFore      (const ColorSpec *c);
pascal void SaveBack         (ColorSpec *c);
pascal void RestoreBack      (const ColorSpec *c);

```

Animating Color Tables

```

pascal void AnimateEntry      (WindowPtr dstWindow,short dstEntry,const RGBColor
                               *srcRGB);

pascal void AnimatePalette    (WindowPtr dstWindow,CTabHandle srcCTab,short
                               srcIndex,short dstEntry,short dstLength);

```

Manipulating Palettes and Color Tables

```

pascal void CopyPalette       (PaletteHandle srcPalette,PaletteHandle
                               dstPalette,short srcEntry,short dstEntry,short
                               dstLength);

pascal void ResizePalette     (PaletteHandle p,short size);

pascal void RestoreDeviceClut (GDHandle gd);

pascal void CTabToPalette     (CTabHandle srcCTab,PaletteHandle dstPalette,short
                               srcUsage,short srcTolerance);

pascal void PaletteToCTab     (PaletteHandle srcPalette,CTabHandle dstCTab);

```

Manipulating Palette Entries

```

pascal void GetEntryColor     (PaletteHandle srcPalette,short srcEntry,RGBColor
                               *dstRGB);

pascal void SetEntryColor     (PaletteHandle dstPalette,short dstEntry,const
                               RGBColor *srcRGB);

pascal void GetEntryUsage     (PaletteHandle srcPalette,short srcEntry,short
                               *dstUsage,short *dstTolerance);

```



```
pascal void SetEntryUsage      (PaletteHandle dstPalette,short dstEntry,short
                               srcUsage,short srcTolerance);

pascal long Entry2Index       (short entry);
```


Color Picker Manager

Contents

About the Color Picker Manager	2-3
Color Picker Dialog Boxes	2-4
Color Pickers as Components	2-6
ColorSync Colors and the Color Picker Manager	2-7
Using the Color Picker Manager	2-8
Using the Standard Dialog Box for Color Pickers	2-9
Defining an Event Filter Function	2-10
Defining a Color-Changed Function	2-11
Using Customized Dialog Boxes for Color Pickers	2-13
Creating Dialog Boxes for Color Pickers	2-14
Setting Colors for and Getting Colors From the Color Picker	2-19
Handling Events in a Color Picker Dialog Box	2-21
Handling Events in the Edit Menu	2-24
Sending Event Forecasters to the Color Picker	2-27
Setting the Destination Profile	2-28
Controlling the Help Balloons for a Color Picker's Dialog Box	2-29
Writing Your Own Color Pickers	2-31
Creating a Component Resource for a Color Picker	2-32
Dispatching to Functions Defined by a Color Picker	2-34
Initializing Your Color Picker	2-37
Handling Events for Your Color Picker	2-41
Returning and Setting Color Picker Information	2-44
Summary of the Color Picker Manager	2-51
Constants and Data Types	2-51
Color Picker Manager Functions	2-57
Application-Defined Functions	2-60
Color Picker-Defined Functions	2-60

This chapter describes how your application can use the standard user interface for soliciting color choices from users. The **Color Picker Manager** supplies the functions that provide your application with this user interface. Your application can use these functions to display, respond to events within, and close a color picker dialog box. The Color Picker Manager provides standard **color pickers** (which allow users to select colors from ranges of colors) and a standard dialog box allowing users to interact with the color picker. This chapter also describes how your application can create its own color pickers and dialog boxes.

The Color Picker Manager version 2.0, described in this chapter, is available on all Macintosh computers using QuickDraw GX or system software version 7.5. Previous versions of system software supported an earlier version of the Color Picker Manager called the Color Picker Package. The Color Picker Manager is compatible with the older Color Picker Package.

Read this chapter if your application allows users to pick a color from a range of colors.

The Color Picker Manager supports ColorSync 1.0. For information about ColorSync 1.0, see the appendix “ColorSync Manager Backward Compatibility” in this book. Because the Color Picker Manager presents color pickers in a dialog box, you should be familiar with the information described in the chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*. The Color Picker Manager uses the Component Manager to interact with color pickers; if you want to create your own color picker, you should be familiar with the information in the chapter “Component Manager” in *Inside Macintosh: More Macintosh Toolbox*.

About the Color Picker Manager

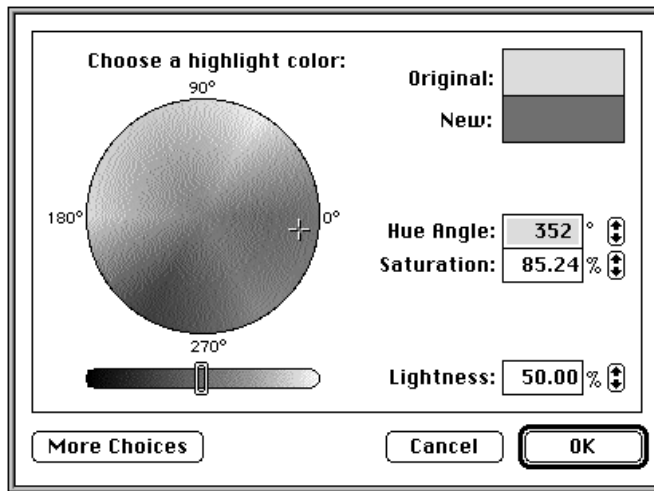
The Color Picker Manager provides your application with a standard way of asking the user to make a color choice.

For users, system software provides standard color pickers appearing in a dialog box that can be modified by your application. Users add new color pickers by installing them in the Extensions folder in the System Folder. The Color Picker Manager also gives you the capability to create color pickers of your own design.

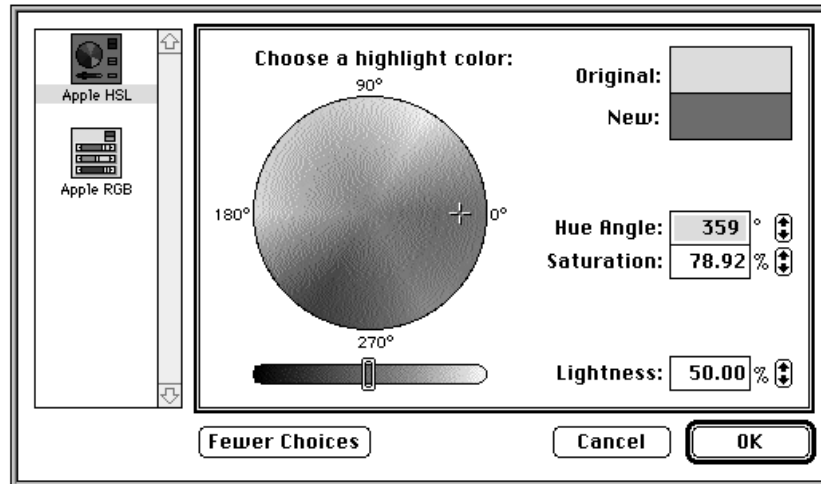
Color Picker Dialog Boxes

If your application uses the standard dialog box for color pickers, your application's interaction with the Color Picker Manager is straightforward: your application calls the `PickColor` function, and the Color Picker Manager presents a standard dialog box to the user, as shown in Figure 2-1. (The older Color Picker Package function `GetColor` also displays this dialog box.)

Figure 2-1 The standard dialog box for color pickers

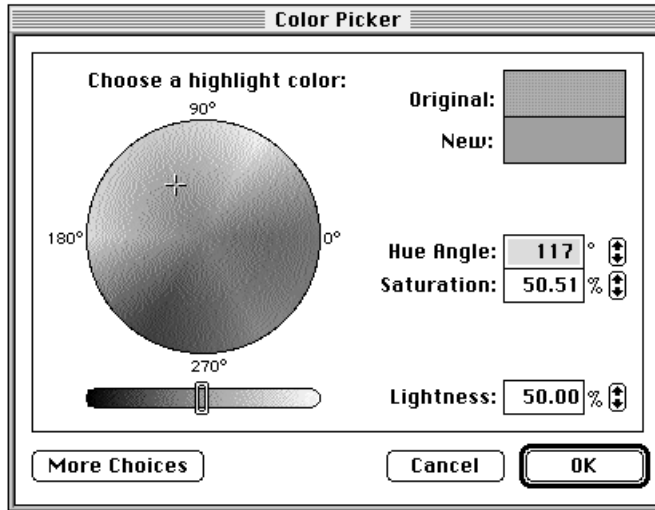


If the user clicks the More Choices button in this dialog box, the Color Picker Manager displays in a scrolling list all available color pickers, as shown in Figure 2-2.

Figure 2-2 Color picker choices in the standard dialog box for color pickers

The user manipulates the controls in this dialog box to select a color. When the user is satisfied with a color and clicks the OK button, `PickColor` returns the selected color to your application and closes the dialog box. The `PickColor` function also closes the dialog box when the user clicks the Cancel button.

By using low-level Color Picker Manager functions, your application can use moveable modal or modeless dialog boxes instead of the modal dialog box displayed by `PickColor`. Figure 2-3 shows a moveable modal dialog box for color pickers.

Figure 2-3 A movable modal dialog box for color pickers

Color Pickers as Components

Color pickers are implemented as *components*, which, as described in the chapter “Component Manager” in *Inside Macintosh: More Macintosh Toolbox*, are pieces of code that provide a set of services to one or more clients. For color pickers, the Color Picker Manager is the client. When your application calls a Color Picker Manager function, it calls the Component Manager, which in turn calls the color picker itself.

The separation of color pickers from the Color Picker Manager allows the user or your application to dynamically add new color pickers to the user’s system. Once a new color picker has been registered with the Component Manager, it’s available for use by the Color Picker Manager. The section “Writing Your Own Color Pickers,” beginning on page 2-31 describes how to create your own color picker. However, most applications do not need to create their own pickers, because those supplied with system software are usually sufficient.

ColorSync Colors and the Color Picker Manager

When returning a color selected by the user, the Color Picker Manager uses the ColorSync definition of a color, which contains both a color and a color-matching profile. The color-matching profile defines the color space of the color (which includes the type of color — CMYK, HSL, RGB, and so on). Your application can also specify a destination profile, which describes the color space of the device for which the color is being chosen (for example, a color printer that will eventually print the document). When given the destination profile, color pickers that are ColorSync aware can help the user choose a color that's within the gamut of the destination device.

This version of the Color Picker Manager uses ColorSync 1.0 profiles only. The ColorSync 1.0 profile is a handle-based profile. The profile format is defined by Apple Computer. You cannot use version 2.0 profiles, which are identified by profile references, with this version of the Color Picker Manager. ColorSync 1.0 profiles typically reside in the ColorSync™ Profiles folder (within the Preferences folder of the System Folder). They may also be embedded with the images to which they pertain in graphics files. The appendix “ColorSync Manager Backward Compatibility” provides information about the relationship between the ColorSync Manager version 2.0 and ColorSync 1.0 profiles, which you may find useful. Because ColorSync 1.0 is supported for backward compatibility only, the ColorSync 1.0 profile format is not described in this book.

For compatibility with the Color Picker Package, the `GetColor` function still uses RGB colors. To convert RGB colors to and from those of the CMYK, HSL, and HSV models, you can use the functions described in “Converting Colors Among Color Models” in the chapter “Color Picker Manager Reference” in *Advanced Color Imaging Reference* on the enclosed CD. See the chapter “Introduction to the ColorSync Manager” in this book for an explanation of these color models.

Using the Color Picker Manager

You can use the Color Picker Manager to allow the user to select a color. The functions defined by the Color Picker Manager are divided into one high-level function and several low-level functions:

- The high-level function, `PickColor`, provides access to almost all of the Color Picker Manager's feature set. For compatibility with the older Color Picker Package, its old high-level function, `GetColor`, is still supported by the Color Picker Manager. The use of `PickColor` is described in "Using the Standard Dialog Box for Color Pickers," beginning on page 2-9.
- The low-level functions allow maximum flexibility for your application. The `PickColor` function presents a modal dialog box, but the low-level functions allow your application to use moveable modal and modeless dialog boxes. These low-level functions are described in "Using Customized Dialog Boxes for Color Pickers," beginning on page 2-13.

Before calling the Color Picker Manager functions, your application should test for the availability of the Color Picker Manager by calling the `Gestalt` function with the `gestaltColorPickerVersion` selector.

```
enum {
    gestaltColorPickerVersion = 'cpkr'    /* returns version of the
                                           Color Picker Manager */
};
```

If the `Gestalt` function returns a value of 00000200, version 2.0 of the Color Picker Manager is available. If the `Gestalt` function returns a value of 00000100, version 1.0 (that is, the original Color Picker Package) is available.

Besides using the Color Picker Manager to interact with color pickers already available on the user's system, you can use the Component Manager to create your own color picker, as described in "Writing Your Own Color Pickers," beginning on page 2-31. Other applications can then interact with your color picker by using the Color Picker Manager.

Using the Standard Dialog Box for Color Pickers

When your application calls the `PickColor` function, your application passes it a pointer to a color picker parameter block. Your application uses the color picker parameter block (which is described in detail in the chapter “Color Picker Manager Reference” in *Advanced Color Imaging Reference* on the enclosed CD) to specify information to and obtain information from the Color Picker Manager. For example, Listing 2-1 uses the color picker parameter block to specify which color to initially display in the dialog box and to place the dialog box on the deepest color screen. You also use the color picker parameter block to determine whether the user has selected a new color, and, if so, which color is selected.

Listing 2-1 Using the `PickColor` function

```
void MyPickAColor(void)
{
    ColorPickerInfo cpInfo;
    /* setting input color to be an RGB color */
    cpInfo.theColor.color.rgb = gMyRGBColor;
    cpInfo.theColor.profile = 0L;
    /* no destination profile */
    cpInfo.dstProfile = 0L;
    cpInfo.flags = AppIsColorSyncAware | CanModifyPalette |
                  CanAnimatePalette;
    /* center dialog box on the deepest color screen */
    cpInfo.placeWhere = kDeepestColorScreen;
    /* use the system default picker */
    cpInfo.pickerType = 0L;
    /* install event filter and color-changed functions */
    cpInfo.eventProc = MyEventProc; /* see Listing 2-2 on page 2-11 */
    cpInfo.colorProc = MyColorChangedProc; /* see Listing 2-3 */
    cpInfo.colorProcData = 0L;
    strcpy(cpInfo.prompt, "\pChoose a highlight color:");
    /* describe the Edit menu for Color Picker Manager */
    cpInfo.mInfo.editMenuID = kMyEditMenuID;
    cpInfo.mInfo.cutItem = kMyCutItem;
    cpInfo.mInfo.copyItem = kMyCopyItem;
    cpInfo.mInfo.pasteItem = kMyPasteItem;
    cpInfo.mInfo.clearItem = kMyClearItem;
```

Color Picker Manager

```

    cpInfo.mInfo.undoItem = kMyUndoItem;
    /* display dialog box to allow user to choose a color */
    if(PickColor(&cpInfo) == noErr && cpInfo.newColorChosen)
        /* use this new color */
        DoNewColor(&cpInfo.theColor);
}

```

The `PickColor` function displays the modal dialog box shown in Figure 2-1 on page 2-4. Your application uses the `prompt` field of the color picker parameter block to specify the text string prompting the user to choose a color for a particular use (for example, “Choose a highlight color”).

When the user clicks the Cancel button, `PickColor` removes the dialog box, and the `newColorChosen` field of the color picker parameter block contains the value `false`. When the user clicks the OK button, `PickColor` removes the dialog box, the `newColorChosen` field of the color picker parameter block contains the value `true`, and the field `theColor` contains a structure describing the newly selected color. Your application can then use this color for the purpose described in the prompt string (for example, highlighting text or filling shapes).

When the dialog box is first displayed, the color that your application specifies in the field `theColor` is used as the original color from which the user begins to edit. Upon completion, `PickColor` sets this field to the last color that the user chose before clicking OK. Although the new colors selected by the user may vary widely, the original color remains fixed for comparison. Figure 2-1 on page 2-4 shows how the standard dialog box displays both the original and the new colors.

Defining an Event Filter Function

Applications can generally allow the Color Picker Manager to handle all events that might occur while the standard dialog box is displayed. Update events are exceptions to this, however.

The `PickColor` function calls the Dialog Manager function `DialogSelect`. As described in the chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*, `DialogSelect` does not allow background windows to receive update events; therefore, at a minimum, your event filter function should handle update events. If your application needs to filter or preprocess other events before `DialogSelect` handles them, your application should do so in its event filter function.

As you can see in Listing 2-1 on page 2-9, your application should supply the `eventProc` field of the color picker parameter block with a pointer to an application-defined filter function for handling user events meant for your application. If your filter function returns `true`, the Color Picker Manager won't process the event any further. If your filter function returns `false`, the Color Picker Manager handles the event as if it were meant for the color picker. Listing 2-2 illustrates such an event filter function.

Listing 2-2 An event filter function for the `PickColor` function

```
pascal Boolean MyPickerEventFilterFunction (EventRecord *event)
{
    /* returning false sends events to the Color Picker Manager */
    Boolean handled = false;
    switch(event->what)
    {
        case updateEvt:
            DoTheUpdate((WindowPtr) event->message);
            handled = true;
    }
    return handled;
}
```

Defining a Color-Changed Function

As shown in Listing 2-1 on page 2-9, your application can supply the `colorProc` field of the color picker parameter block with a pointer to an application-defined function that handles color changes. This function, illustrated in Listing 2-3, should support the updating of colors in a document as the user selects them. To support faster feedback as the user changes color, this function doesn't update the application's internal data structures.

Listing 2-3 A color-changed function

```
pascal void MyColorChangedFunction(long userData, PMColorPtr newColor)
{
    GrafPtr    port;
    CWorld     cWorld;
    CMColor    color;
    CMError    cWError;

    GetPort(&port);
    SetPort(myDocWindow);
    /* determine whether the color has a profile */
    if(newColor->profile)
    {
        /* convert the color to an RGB color */
        cWError = CWorldNewColorWorld(&cWorld, newColor->profile, 0L);
        if(cWError == noErr || cWError == CMProfilesIdentical)
        {
            color = newColor->color;
            CWorldMatchColors(cWorld, &color, 1);
            CWorldDisposeColorWorld(cWorld);
        }
    }
    else
        color.rgb = newColor->color.rgb;
    /* change the color of currently highlighted objects */
    MyChangeHighlight(&color.rgb);
    SetPort(port);
}
```

In this example, it is assumed that ColorSync is installed because the application sets the `AppIsColorSyncAware` flag when calling `PickColor`. Because a non-RGB color might come back from the color picker, this example uses ColorSync 1.0 functions to convert the color to an RGB color.

Using Customized Dialog Boxes for Color Pickers

Instead of using the `PickColor` function, your application can use the low-level Color Picker Manager functions to create a tighter integration between your application and the dialog box for color pickers. For example, your application can create a floating palette instead of a modal dialog box. When using the low-level Color Picker Manager functions, your application specifies the type of dialog box in which to put the color picker, and your application maintains tighter control over the event loop.

Your application can use the low-level calls to create three additional types of color picker dialog boxes: system-owned, application-owned, and color picker-owned.

A **system-owned dialog box** for color pickers has the same dialog box items as the dialog box created by `PickColor`—that is, it has OK, Cancel, and More Choices buttons. However, with the low-level calls, you can make the dialog box a movable modal or modeless dialog box.

Application-owned dialog boxes are supplied by applications. You can use this type of dialog box to integrate color pickers with other window features of your application or to extend the controls for color pickers. For example, you could add controls that allow the user to alter the style of an object as well as its color.

A **color picker-owned dialog box** is created by the color picker itself. Creating its own dialog box gives a color picker the flexibility of specifying the size and shape of the color picker (color pickers in system-owned and application-owned dialog boxes are always the same size). This is useful for implementing color pickers in floating palettes.

Once they're created, your application interacts with all three types of dialog boxes in the same way. The rest of this section describes how to create each type of dialog box and then discusses how your application interacts with the color picker displayed in the dialog box, no matter what type of dialog box you've used.

Creating Dialog Boxes for Color Pickers

The code for creating all three dialog boxes is similar. In all three cases, your application

- Creates a structure describing the dialog box and your application's Edit menu. For a system-owned dialog box, this is a `SystemDialogInfo` structure; for an application-owned dialog box, this is an `ApplicationDialogInfo` structure; for a color picker-owned dialog box, this is a `PickerDialogInfo` structure. These structures are described in detail in the chapter "Color Picker Manager Reference" in *Advanced Color Imaging Reference* on the enclosed CD.
- Uses the `SetPickerColor` function (described in *Advanced Color Imaging Reference* on the enclosed CD) to set the original and new colors for the color picker.
- Uses the `SetPickerPrompt` function (described in *Advanced Color Imaging Reference* on the enclosed CD) to specify a text string prompting the user to choose a color for a particular use (for example, "Choose a highlight color").
- Uses—for an application-owned dialog box—the `AddPickerToDialog` function (described in *Advanced Color Imaging Reference* on the enclosed CD) to add a color picker to your application's dialog box.
- Makes the dialog box visible (if your application initially created it as invisible) with the `SetPickerVisibility` function (described in *Advanced Color Imaging Reference*) and, for an application-owned dialog box, with the Window Manager function `ShowWindow` (described in the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.)

When creating a system-owned dialog box, your application uses combinations of the `DialogIsModal` and `DialogIsMoveable` flags in the `flags` field of a `SystemDialogInfo` structure to specify whether the dialog box is modal, movable modal, or modeless. (You should always set one or both flags, because your application should never display a nonmovable, modeless dialog.)

Listing 2-4 illustrates how to create the movable modal system-owned dialog box shown in Figure 2-3 on page 2-6.

Listing 2-4 Creating a movable modal system-owned dialog box

```

OSErr MyBuildMovableModalSysDialog(void)
{
    SystemDialogInfo    sInfo;
    OSErr               result;

    sInfo.flags = DialogIsMoveable + AppIsColorSyncAware +
                  CanModifyPalette + CanAnimatePalette;
    sInfo.pickerType = 0L;
    sInfo.placeWhere = kDeepestColorScreen;
    sInfo.mInfo.editMenuID = kMyEditMenuID;
    sInfo.mInfo.cutItem = kMyCutItem;
    sInfo.mInfo.copyItem = kMyCopyItem;
    sInfo.mInfo.pasteItem = kMyPasteItem;
    sInfo.mInfo.clearItem = kMyClearItem;
    sInfo.mInfo.undoItem = kMyUndoItem;
    gMyPicker = nil;
    result = CreateColorDialog(&sInfo, &gMyPicker);
    if(result == noErr && gMyPicker != nil)
    {
        PMColor myPMColor;
        myPMColor.color.rgb = gMyRGBColor;
        myPMColor.profile = 0L;
        SetPickerColor(gMyPicker, kOriginalColor, &myPMColor);
        SetPickerColor(gMyPicker, kNewColor, &myPMColor);
        SetPickerPrompt(gMyPicker, "\pChoose a highlight color:");
        SetPickerVisibility(gMyPicker, true);
    }
    return result;
}

```

Before displaying the dialog box, your application must use the `SetPickerColor` function to specify colors for the user to start with. Setting these *original* and *new* colors is described in the next section.

Listing 2-5 shows how to use the Dialog Manager function `GetNewDialog` to create an application-owned dialog box. (See the chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials* for information on the `GetNewDialog` function.) To use this dialog box for a color picker, your

Color Picker Manager

application needs to create an `ApplicationDialogInfo` structure and then call the `AddPickerToDialog` function, as shown in this example.

Listing 2-5 Creating an application-owned dialog box

```
OSErr MyBuildAppDialog(void)
{
    ApplicationDialogInfo  aInfo;
    OSErr                  result;

    /* create the dialog box, but ensure it's color */
    gMyDialog = GetNewDialog(kMyDialogID, nil, (WindowPtr)-1);
    /* set up the ApplicationDialogInfo structure */
    aInfo.flags = DialogIsMoveable + AppIsColorSyncAware +
                  CanModifyPalette + CanAnimatePalette;
    aInfo.pickerType = 0L;
    aInfo.theDialog = gMyDialog;
    /* put the color picker's origin at (0,0) in the dialog box */
    aInfo.pickerOrigin.h = 0;
    aInfo.pickerOrigin.v = 0;
    /* report Edit menu information */
    aInfo.mInfo.editMenuID = kMyEditMenuID;
    aInfo.mInfo.cutItem = kMyCutItem;
    aInfo.mInfo.copyItem = kMyCopyItem;
    aInfo.mInfo.pasteItem = kMyPasteItem;
    aInfo.mInfo.clearItem = kMyClearItem;
    aInfo.mInfo.undoItem = kMyUndoItem;
    /* add the color picker to the dialog box */
    result = AddPickerToDialog(&aInfo, &gMyPicker);
    if(result == noErr && gMyPicker != nil)
    {
        PMColor myPMColor;
        myPMColor.color.rgb = gMyRGBColor;
        myPMColor.profile = 0L;
        SetPickerColor(gMyPicker, kOriginalColor, &myPMColor);
        SetPickerColor(gMyPicker, kNewColor, &myPMColor);
        SetPickerPrompt(gMyPicker, "\npChoose a highlight color");
        SetPickerVisibility(gMyPicker, true);
        ShowWindow(gMyDialog);
        DrawDialog(gMyDialog);
    }
}
```

Color Picker Manager

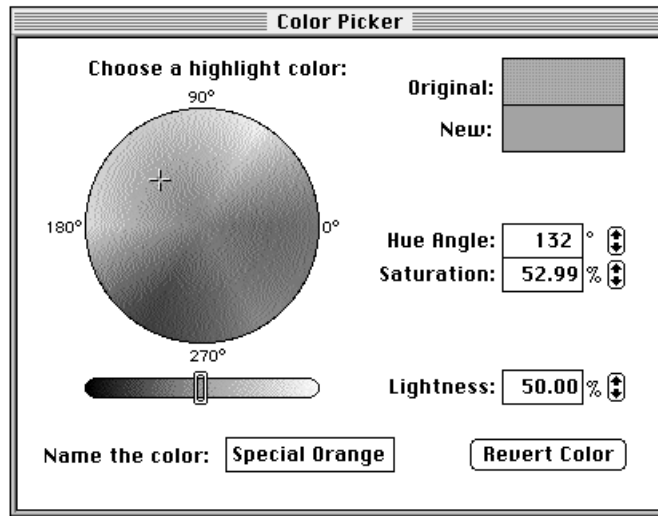
```

    }
    else
        MyDoError(result);
    return result;
}

```

Figure 2-4 shows the application-owned dialog box created by Listing 2-5. Notice the difference between the buttons in this dialog box and those in the system-owned dialog box shown in Figure 2-3 on page 2-6.

Figure 2-4 A movable modal application-owned dialog box



Listing 2-6 shows how use a `PickerDialogInfo` structure and the `CreatePickerDialog` function to create a color picker-owned dialog box.

Listing 2-6 Creating a color picker–owned dialog box

```

OSErr BuildPickerDialog(void)
{
    PickerDialogInfo    pInfo;
    OSErr                result;

    pInfo.flags = DialogIsMoveable + AppIsColorSyncAware +
                  CanModifyPalette + CanAnimatePalette;
    pInfo.pickerType = 0L;
    pInfo.mInfo.editMenuID = kMyEditMenuID;
    pInfo.mInfo.cutItem = kMyCutItem;
    pInfo.mInfo.copyItem = kMyCopyItem;
    pInfo.mInfo.pasteItem = kMyPasteItem;
    pInfo.mInfo.clearItem = kMyClearItem;
    pInfo.mInfo.undoItem = kMyUndoItem;
    gMyPicker = nil;
    result = CreatePickerDialog(&pInfo, &gMyPicker);
    if(result == noErr && gMyPicker != nil)
    {
        PMColor myPMColor;
        myPMColor.color.rgb = gMyRGBColor;
        myPMColor.profile = 0L;
        SetPickerColor(gMyPicker, kOriginalColor, &myPMColor);
        SetPickerColor(gMyPicker, kNewColor, &myPMColor);
        SetPickerPrompt(gMyPicker, "\pChoose a highlight color:");
        SetPickerVisibility(gMyPicker, true);
    }
    else
        MyDoError(result);
    return result;
}

```

As you can see from Listing 2-4, Listing 2-5, and Listing 2-6, the code to create all three types of dialog boxes is very similar. As you will see in the next few sections, the code to manage them is also very similar.

Setting Colors for and Getting Colors From the Color Picker

Your application uses a color picker to present color choices to the user and to determine which color the user selects.

Most color pickers present controls that identify color choices. In any case, your application must initially set two default colors: an original color and a new color. The **original color** is the color that the user is about to change, and the **new color** is the color to which the user changes the original. For custom color picker dialog boxes, you use the `SetPickerColor` function to set both colors. (When your application uses the `PickColor` function to display the standard dialog box, your application supplies the original color in the field `theColor` of the color picker parameter block. This color is used as the initial value for both the original color and the new color.)

Although the new colors selected by the user may vary widely, the original color remains fixed for comparison. Figure 2-3 on page 2-6 shows how a system-owned dialog box displays both the original and the new colors.

Suppose, for example, that your paint program uses a floating palette for a color picker. When the user clicks an object, you want the floating palette to show the color of that object. You accomplish this by initially setting the object's current color as both the original color and the new color. As the user changes the color of the object, the original color remains the same while the new color changes. This provides feedback as to what would happen if the user were to undo the color change.

Listing 2-7 illustrates how to use the `SetPickerColor` function to set the original and new colors.

Listing 2-7 Setting the original and new colors

```
void MySetPickerToColor(RGBColor *rgb)
{
    PMColor aColor;

    aColor.color.rgb = *rgb;
    aColor.profile = 0L;
    SetPickerColor(myPicker, kOriginalColor, &aColor);
    SetPickerColor(myPicker, kNewColor, &aColor);
}
```

Color Picker Manager

Whenever the user changes the current color, you need to be able to get the new color so that you can update your object accordingly. To determine what color the user is selecting, use the `GetPickerColor` function, as illustrated in Listing 2-8. (Use the `GetPickerColor` function for getting colors from color pickers in custom dialog boxes. When your application uses the `PickColor` function to display the standard dialog box, and the user clicks the OK button, the Color Picker Manager returns the new color in the field `theColor` of the color picker parameter block.)

Listing 2-8 Determining the selected color

```
void MyGetCurrentColor(RGBColor *rgb)
{
    PMColor aColor;

    GetPickerColor(myPicker, kNewColor, &aColor);
    *rgb = aColor.color.rgb;
}
```

As shown in Listing 2-3 on page 2-12, you might want to use the colors provided by your color-changed function as temporary colors only. Accordingly, your application should not update its internal data until the user has actually chosen a color (or at least stopped dragging a control). Your application will know this happens when the `DoPickerEvent` function, which is described in the next section, returns the `kColorChanged` constant. Your application should then update its internal data.

As you can see, setting colors for and getting colors from a color picker are simple tasks to perform; complexities arise only if you need to convert a color returned by the color picker from one color space to another. For example, a color picker might return a color using the CMYK color space; if your application uses only RGB colors, then your application must convert the color between the two spaces. (See “Converting Colors Among Color Models” in the chapter “Color Picker Manager Reference” in *Advanced Color Imaging Reference* on the enclosed CD for information about converting between RGB colors and other color types.)

Listing 2-3 on page 2-12 illustrates how to use ColorSync 1.0 functions to convert any color to an RGB color.

When the `AppIsColorSyncAware` flag is not set in the `SystemDialogInfo`, `ApplicationDialogInfo`, or `PickerDialogInfo` structure used to create a dialog box, the Color Picker Manager automatically converts any color it gets back from a color picker into an RGB color.

Handling Events in a Color Picker Dialog Box

When your application receives an event for a color picker dialog box, you use the `DoPickerEvent` function to pass the event to the Color Picker Manager. The color picker or the Color Picker Manager either handles the event or returns it to your application for handling.

When your application uses the `DoPickerEvent` function to pass an event to a color picker or the Color Picker Manager for handling, you use an `EventData` structure to supply information about the event, and to receive information about how the color picker or the Color Picker Manager handled the event. (The `DoPickerEvent` function and the `EventData` structure are described in detail in the chapter “Color Picker Manager Reference” in *Advanced Color Imaging Reference* on the enclosed CD.)

When either the color picker or the Color Picker Manager handles the event, the `DoPickerEvent` function returns in the `action` field of the `EventData` structure one of the following constants describing the event. (These constants are described in “Picker Actions” in the chapter “Color Picker Manager Reference” in *Advanced Color Imaging Reference* on the enclosed CD.)

```
enum PickerAction {
    kDidNothing,          /* no action worth reporting */
    kColorChanged,        /* user chose a different color */
    kOkHit,               /* user clicked OK */
    kCancelHit,           /* user clicked Cancel */
    kNewPickerChosen,      /* user chose a new color picker */
    kAppItemHit           /* Dialog Manager returned an item in an
                           application-owned dialog box */
};

typedef short PickerAction;
```

Internally, the Color Picker Manager handles the event by calling the `DialogManager` function `DialogSelect` and then processing the event from there. If the color picker is in an application-owned dialog box and an application item is selected, the Color Picker Manager returns the `kAppItemHit` constant as well as the number of the selected item.

Color Picker Manager

If your application creates a modeless dialog box for a color picker, you must handle events related to its menus before calling the `DoPickerEvent` function. If you've created a modal system-owned dialog box, the Color Picker Manager can handle the Edit menu events for you (as it does when you call `PickColor`). However, for other dialog boxes there may be menu items that the Color Picker Manager cannot handle. If you send events relating to these menu items to the Color Picker Manager, it will assume all Edit menu selections are meant for the color picker and will ignore every other menu selection. "Handling Events in the Edit Menu" on page 2-24 describes how to handle the Edit menu.

Listing 2-9 illustrates an event loop. This example assumes that the application always handles events related to its menus.

Listing 2-9 A sample event loop

```
#define IsMenuKey(x) (x)->what == keyDown && (x)->modifiers &
                                                                    cmdKey)

Boolean MySampleDoEvent(EventRecord *event)
{
    Boolean        handled = false, isMenuEvent = false;
    EventData      pEvent;
    short          inWhere;
    WindowPtr      whichWindow;

    if (event->what == mouseDown)
    {
        inWhere = FindWindow(event->where, &whichWindow);
        if (inWhere == inMenuBar)
            isMenuEvent = true;
    }
    if (isMenuEvent || IsMenuKey(event))
    {
        DoMenu(event);
        handled = true;
    }
    /* if the event's not handled yet, pass it to the Color Picker
       Manager */
    if (!handled)
    {
        pEvent.event = event;
```


Color Picker Manager

```

pEvent.colorProc = MyColorChangedProc;
pEvent.colorProcData = 0L;
DoPickerEvent(myPicker, &pEvent);
handled = pEvent.handled;
/* if the color picker handled it, do something with the
   results */
if (handled)
{
    switch (pEvent.action) {
        case kDidNothing:
            break;
        case kColorChanged:
            UseNewColor(myPicker);
            break;
        case kOKHit:
            UseNewColor(myPicker);
            DisposeColorPicker(myPicker);
            myPicker = nil;
            break;
        case kCancelHit:
            UseOriginalColor(myPicker);
            DisposeColorPicker(myPicker);
            myPicker = nil;
            break;
        case kNewPickerChosen:
            /* nothing to do for this case */
            break;
        case kApplItemHit:
            /* let my app handle the item */
            MyHandleAppItem(pEvent.itemHit);
            break;
    }
}
if (!handled)
{
    /* the event hasn't been handled; treat it like any normal
       Macintosh event; if any other dialog boxes are present,
       call DialogSelect here; if the event is a
       mouseDown event, my app already called FindWindow */

```

Color Picker Manager

```

    }
    return handled;
}

```

As shown in this example, if you use a color-changed function (such as the one illustrated in Listing 2-3 on page 2-12), you should supply it, along with any data it needs, in the `EventData` structure that your application passes to the `DoPickerEvent` function.

Handling Events in the Edit Menu

Handling events in the Edit menu requires more work than standard menu processing. If an Edit menu choice is for the color picker, you need to set the state of the Edit menu items according to the color picker specifications and, if a menu item is chosen, send the appropriate message to the color picker. To do so, you use two functions, `GetPickerEditMenuState` and `DoPickerEdit`.

After you determine that there has been a mouse-down event in the Edit menu (or that the user pressed a keyboard equivalent), you need to determine who owns the Edit menu.

If the color picker is in a color picker-owned or system-owned dialog box and it's the frontmost window, the color picker owns it. If the color picker is in an application-owned dialog box and it's frontmost, ownership of the Edit menu depends on the current dialog item. The choice really depends on your application. As a general rule, whoever owns the current item owns the Edit menu. If your application uses the `DoPickerEdit` function while the current item belongs to your application, `DoPickerEdit` implements the standard cut, copy, paste, and clear features for your application. If your application needs to do more than this, it needs to handle the menu itself.

Listing 2-10 assumes that the owner of the current item owns the Edit menu. The item number for the application's last dialog item is represented by the constant `kMyLastItem`. If your application has a system-owned or color picker-owned dialog box, this constant should be set to 0. In an application-owned dialog box, the color picker's items will always be added after your application's, so your item numbers remain the same.

Listing 2-10 Handling the Edit menu

```

Boolean MyDoMenu(EventRecord *event)
{
    long          mChoice;
    EditData      eData;
    EditOperation eOperation;

    /* if the picker is in front and the current edit item is the
       picker's, set up the Edit menu as the picker wants it */
    if (FrontWindow() == gMyDialog &&
        ((DialogPeek)gMyDialog)->editField + 1 > kMyLastItem)
    {
        MenuState      mState;
        MenuHandle      theMenu;

        GetPickerEditMenuState(myPicker, &mState);
        theMenu = GetMenu(kMyEditMenuID);
        if (mState.cutEnabled)
            EnableItem(theMenu, kMyCutItem);
        else
            DisableItem(theMenu, kMyCutItem);
        if (mState.copyEnabled)
            EnableItem(theMenu, kMyCopyItem);
        else
            DisableItem(theMenu, kMyCopyItem);
        if (mState.pasteEnabled)
            EnableItem(theMenu, kMyPasteItem);
        else
            DisableItem(theMenu, kMyPasteItem);
        if (mState.clearEnabled)
            EnableItem(theMenu, kMyClearItem);
        else
            DisableItem(theMenu, kMyClearItem);
        if (mState.undoEnabled)
        {
            SetItem(theMenu, kMyUndoItem, mState.undoString);
            EnableItem(theMenu, kMyUndoItem);
        }
        else
            DisableItem(theMenu, kMyUndoItem);
    }
}

```

Color Picker Manager

```

    }
    /* pass the event to the Menu Manager */
    if (event->what == mouseDown)
        mChoice = MenuSelect(event->where);
    else if (event-> == keyDown)
        mChoice = MenuKey(event->message);
    /* if not the Edit menu, handle normally */
    if (HiWord(mChoice) != kMyEditMenuID)
    {
        HandleMenuChoice(mChoice);
        return true;
    }
    switch (LoWord(mChoice))
    {
        case kMyCutItem:
            eOperation = kCut;
            break;
        case kMyCopyItem:
            eOperation = kCopy;
            break;
        case kMyPasteItem:
            eOperation = kPaste;
            break;
        case kMyClearItem:
            eOperation = kClear;
            break;
        case kMyUndoItem:
            eOperation = kUndo;
            break;
        default:
            eOperation = -1;
            break;
    }
    if (eOperation >= 0)
    {
        eData.theEdit = eOperation;
        DoPickerEdit(myPicker, &eData);
        /* this example is simply ignoring the results here */
    }

```

Color Picker Manager

```

        HiliteMenu(0);
        return true;
    }

```

See the chapter “Menu Manager” in *Inside Macintosh: Macintosh Toolbox Essentials* for more information about managing menus in your application.

Sending Event Forecasters to the Color Picker

Your application may sometimes need to warn the color picker about a user action that might affect it. For example, if you display a color picker in an application-owned dialog box and the user closes that dialog box, your application might want to check whether the color picker is in a state that can handle this. If the user had just typed some numbers into the color picker that left it in an inconsistent state, it would be helpful if the color picker could warn the user before the user closes it.

Your application can warn the color picker about potential problems by sending it **event forecasters**. Event forecasters aren’t events in themselves; instead, they are warnings to the color picker. To send event forecasters to the color picker, you use the same function as for regular events—`DoPickerEvent`—except that in the `EventData` structure that your application passes to `DoPickerEvent`, your application sets the `event` field to `nil` and sets the `forecast` field to an appropriate constant from the following list.

```

enum EventForecasters {
    kNoForecast,          /* no forecast (e.g., an update event) */
    kMenuChoice,          /* this event causes a menu to be chosen */
    kDialogAccept,        /* the dialog box will be accepted */
    kDialogCancel,        /* the dialog box will be canceled */
    kLeaveFocus,           /* the focus will leave the color picker */
    kPickerSwitch,        /* new color picker chosen in More Choices
                           list */
    kNormalKeyDown,       /* a normal key-down event in an edit field */
    kNormalMouseDown      /* a normal click in color picker's focus */
};

typedef short EventForecaster;

```

The color picker informs your application about whether the color picker is ready for the action to occur by setting the `handled` field of the `EventData` structure to `true` if it’s not ready and `false` if it is.

Color Picker Manager

Generally, the only time you need to use event forecasters is when the color picker's dialog box is about to close. If the Color Picker Manager affects the closing (as indicated when it sets the `action` field of the `EventData` structure to the `kOKHit` constant after your application calls `DoPickerEvent`), your application doesn't need to warn the color picker because the Color Picker Manager has already done so. However, if the user has just clicked the close box of a window containing an application-owned dialog box or has chosen Close from a menu, you should send an event forecaster to the color picker.

Listing 2-11 shows an application-defined function called `CheckIfPickerCanClose`. If this function returns true, then the color picker can close; otherwise, it can't close for some reason, and you can assume that the color picker has informed the user of the problem.

Listing 2-11 Warning the color picker that it's about to be closed

```
Boolean MyCheckIfPickerCanClose()
{
    EventData pEvent;
    pEvent.event = 0L;           /* make it a forecast event */
    pEvent.forecast = kDialogAccept;
    DoPickerEvent(myPicker, &pEvent);
    return !pEvent.handled;
}
```

Setting the Destination Profile

If you use a color picker to ask the user for a color intended for an output device, and the device has a color-matching profile, your application can hand this profile to the color picker so that it can communicate the profile's information to the user. You do this with the `SetPickerProfile` function, as illustrated in Listing 2-12. Setting a destination profile is optional; the color picker assumes that there's no profile unless your application uses `SetPickerProfile` to set one.

Listing 2-12 Using the `SetPickerProfile` function to set the destination profile

```
void MySetDestinationProfile(CMProfileHandle profile)
{
    if (SetPickerProfile(myPicker, profile) != noErr)
        MyHandleError();
}
```

There's also a matching function, `GetPickerProfile`, illustrated in Listing 2-13, to get the current destination profile from the color picker.

Listing 2-13 Using the `GetPickerProfile` function to get the destination profile

```
void MyGetDestinationProfile(CMProfileHandle profile)
{
    if (GetPickerProfile(myPicker, profile) != noErr)
        MyHandleError();
}
```

Your application owns the memory of any profiles it gives to or receives from the color picker. When your application sets the destination profile, the color picker makes a copy of the profile handle; when your application gets the destination profile, your application gives the color picker a handle into which it copies the profile data.

Controlling the Help Balloons for a Color Picker's Dialog Box

The Color Picker Manager supports Balloon Help user assistance (which is described in the chapter “Help Manager” in *Inside Macintosh: More Macintosh Toolbox*). Most applications don't need to do anything special to use Balloon Help for a color picker in any type of dialog box. However, if your application needs control over a color picker's help balloon, you can use the `ExtractPickerHelpItem` function to get the help balloon for the color picker. You can then use the Help Manager function `HMShowBalloon` to override the default help balloon and display the altered balloon.

It's up to your application to determine whether the cursor is over a color picker's item or one of your application's items. You can use the Dialog Manager function `FindDialogItem` to determine which item the cursor is over. If

Color Picker Manager

it's over one of your application items, you can put up your own help balloon. Otherwise, use the `ExtractPickerHelpItem` function to get the color picker's balloon, which you can alter or display as it is defined by the color picker. The `ExtractPickerHelpItem` function searches the color picker's help resource for an appropriate balloon. If it can't find one, it returns the `noHelpForItem` result.

Listing 2-14 illustrates how you can control the help balloons for a color picker dialog box. Everything in this example is performed by the Color Picker Manager internally; the example just gives you a general idea of how to use `ExtractPickerHelpItem`.

Listing 2-14 Using the `ExtractPickerHelpItem` function

```
void MyDoPickerBalloonHelp(void)
{
    HelpItemInfo    helpInfo;
    short           itemNo;
    Point           where;
    OSErr           err;

    GetMouse(&where);
    itemNo = FindDialogItem(gMyDialog, where) + 1;
    /* get the color picker's help item */
    helpInfo.options = 0;
    helpInfo.tip.v = helpInfo.tip.h = 0;
    SetRect(&helpInfo.altRect, 0, 0, 0, 0);
    helpInfo.theProc = 0;
    helpInfo.variant = 0;
    helpInfo.helpMessage.hmmHelpType = 0;
    helpInfo.helpMessage.u.hmmPictHandle = 0L;
    err = ExtractPickerHelpItem(myPicker, itemNo, 0, &helpInfo);
    /* show the balloon */
    if (err == noErr)
    {
        /* if altRect is empty, use the item's rectangle */
        if (EmptyRect(&helpInfo.altRect))
        {
            short        iType;
            Handle        iHandle;
            GetDialogItem(gMyDialog, itemNo, &iType, &iHandle,
```



```

                                &helpInfo.altRect);
    }
    /* convert balloon tip's location to local coordinates */
    helpInfo.tip.h += helpInfo.altRect.left;
    helpInfo.tip.v += helpInfo.altRect.top;
    /* convert the balloon tip and the altRect to global
       coordinates */
    LocalToGlobal(&helpInfo.tip);
    LocalToGlobal((Point *) &helpInfo.altRect.top);
    LocalToGlobal((Point *) &helpInfo.altRect.bottom);
    /* show the balloon */
    HShowBalloon (&helpInfo.helpMessage, helpInfo.tip,
                  &helpInfo.altRect, 0L, helpInfo.theProc,
                  helpInfo.variant, kHMRegularWindow);
}
}

```

If your color picker needs to override the help message or another help balloon characteristic for the item specified in the `itemNo` parameter for the `ExtractPickerHelpItem` function, you should do so before using the `Help Manager` function `HShowBalloon` to display the help balloon. Specify the desired help message and characteristics in the `HelpItemInfo` structure pointed to in the `helpInfo` parameter to `HShowBalloon`.

Writing Your Own Color Pickers

Macintosh system software provides color pickers that present ranges of colors from which users can choose particular colors. You can present these color pickers in dialog boxes, as described in “Using the Color Picker Manager,” beginning on page 2-8. This section describes how you can also use the Component Manager and the Color Picker Manager to create your own color pickers, which you implement as components.

When you create a color picker, the Color Picker Manager uses the Component Manager to request services from your color picker. For example, when the user selects a color from your color picker and clicks the OK button in its dialog box, the Color Picker Manager sends your color picker a request to provide the selected color.

Color Picker Manager

You need to read this section only if you are creating a color picker for use in your own application or in those created by others. Before reading this section, you should read the preceding sections of this chapter. For complete details on components and their structure, see the chapter “Component Manager” in *Inside Macintosh: More Macintosh Toolbox*. For complete information about dialog boxes, see the chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*.

Creating a Component Resource for a Color Picker

A color picker is stored as a component resource. It contains a number of resources, including icons, strings, and the standard component resource (a resource of type 'thng') required of any Component Manager component. In addition, a color picker must contain code to handle required request codes passed to it by the Component Manager.

Your color picker must be contained in a resource file. The creator of the file can be any type you wish, but the type of the file must be 'thng'. If your color picker contains a 'BNDL' resource (described in the chapter “Finder Interface” in *Inside Macintosh: Macintosh Toolbox Essentials*), then the file's bundle bit must be set. To allow other applications to use your color picker, you should place it in the Extensions folder (where it will be automatically registered at startup.) Otherwise, you can use the Component Manager function `RegisterComponent` or `RegisterComponentResource` to make your color picker available as your application needs it.

Listing 2-15 shows the Rez listing of a component resource that describes a color picker.

Listing 2-15 A component resource for a color picker

```
resource 'thng' (kPickerID, locked) {
    'cpkr',          /* component type: a color picker */
    'stup',          /* component subtype */
    'wave',          /* color picker manufacturer */
    $00000071,       /* control flags */
    $000000ff,       /* control flags mask */
    'cpkr',          /* resource type for color picker's code */
    kPickerID,       /* resource ID for color picker's code */
    'STR ',          /* resource type for color picker's name */
}
```

Color Picker Manager

```

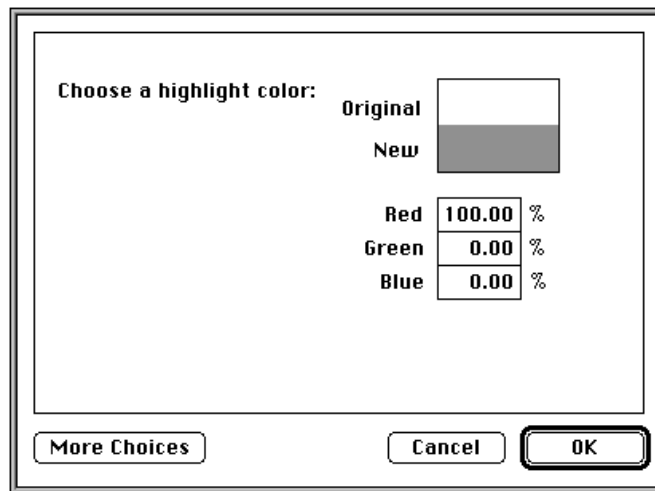
kPickerID,      /* resource ID for color picker's name */
'STR ',        /* color picker info resource type */
kInfoID,        /* color picker info resource ID */
'ICN#',         /* icon list for color picker */
kPickerID      /* icon list resource ID */
};

```

The component resource, and the resources that define the component's code, name, information string, and icon family, must be in the same resource file.

Figure 2-5 shows the color picker created with this component resource. This is a very simplistic color picker that uses only editable text fields for specifying a color.

Figure 2-5 An application-created color picker



Dispatching to Functions Defined by a Color Picker

As explained in the previous section, the code for your color picker should be contained in a resource. The Component Manager expects the entry point in this resource to be a function having this format:

```
pascal ComponentResult MyColorPickerDispatch (
                                ComponentParameters *params,
                                Handle storage)
```

Whenever the Color Picker Manager uses the Component Manager to send a request to your color picker, the Component Manager calls your component's entry point and passes any parameters, along with information about the current connection, in a component parameters structure. The Component Manager also passes a handle to the global storage associated with that instance of your color picker.

When your color picker 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.

The component parameters structure is defined by a data structure of type `ComponentParameters`. The `what` field of this structure specifies the type of request. Your color picker's entry point should interpret the request code and then possibly dispatch to some other subroutine. Your color picker must be able to handle the required request codes represented by the constants `kComponentOpenSelect`, `kComponentCloseSelect`, `kComponentCanDoSelect`, and `kComponentVersionSelect`.

Note

For complete details on required component request codes and the `ComponentParameters` structure, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox*. ♦

Color Picker Manager

In addition to these required request codes, your color picker must also be able to handle the request codes defined by the elements of the `PickerMessages` enumerated data type, which is shown here and described in detail in the chapter “Color Picker Manager Reference” in *Advanced Color Imaging Reference* on the enclosed CD.

```
typedef enum          {          /* request codes handled by a color picker */
    kInitPicker,      /* initialize any private data */
    kTestGraphicsWorld, /* test operability on current system */
    kGetDialog,        /* if using own dialog box, return its pointer; for
                        default dialog box, return nil */
    kGetItemList,      /* return a list of items for dialog box */
    kGetColor,         /* return original or newest color */
    kSetColor,         /* change original or newest color */
    kEvent,            /* perform any special processing necessary for an event */
    kEdit,             /* perform an editing command */
    kSetVisibility,    /* make color picker visible or invisible */
    kDrawPicker,       /* redraw color picker */
    kItemHit,          /* respond to event in a dialog box item */
    kSetBaseItem,      /* set base item for dialog box items */
    kGetProfile,       /* return a handle to the destination profile */
    kSetProfile,       /* change the destination color-matching profile */
    kGetPrompt,        /* return prompt string */
    kSetPrompt,        /* set a new prompt */
    kGetIconData,      /* return script code and resource ID of icon family */
    kGetEditMenuState, /* return information about edit menu */
    kSetOrigin,        /* update any info about local coordinate system */
    kExtractHelpItem   /* return information about help balloons */
} PickerMessages;
```

Your color picker should respond to these request codes by performing the requested action. Your color picker may need to access additional information provided in the `params` field to service the request. The `params` field of the component parameters structure is an array that contains the parameters specified by the Color Picker Manager. Your color picker can use the `CallComponentFunctionWithStorage` function to extract the parameters from this array and pass these parameters to a subroutine of your color picker. Listing 2-16 illustrates how to define a color picker entry point routine. This example inspects the `params->what` field to determine which request code to handle.

Listing 2-16 Handling Component Manager request codes

```

pascal ComponentResult MyColorPickerDispatch(ComponentParameters *params,
                                           Handle storage)
{
    short                message;
    ComponentResult       result;
    ComponentFunction     RoutineToCall = 0;

    message = params->what;
    if (message < 0)
        /* negative values are Component Manager request codes */
        switch (message)
        {
            case kComponentOpenSelect:
                                RoutineToCall = MyOpen; break;
            case kComponentCloseSelect:
                                RoutineToCall = MyClose; break;
            case kComponentCanDoSelect:
                                RoutineToCall = MyCanDo; break;
            case kComponentVersionSelect:
                                RoutineToCall = MyVersion; break;
            case kComponentRegisterSelect:
                                RoutineToCall = MyRegister; break;
            case kComponentTargetSelect:
                                RoutineToCall = MySetTarget; break;
            default:
                                return 0;          /* no error */
        }
    else
    {
        switch (message)
        {
            case kInitPicker:
                                RoutineToCall = MyInitPicker; break;
            case kTestGraphicsWorld:
                                RoutineToCall = MyTestGraphicsWorld; break;
            case kGetDialog:
                                RoutineToCall = MyGetDialog; break;
            case kGetItemList:
                                RoutineToCall = MyGetItemList; break;
            case kGetColor:
                                RoutineToCall = MyGetColor; break;
            case kSetColor:
                                RoutineToCall = MySetColor; break;
            case kEvent:
                                RoutineToCall = MyDoEvent; break;
            case kEdit:
                                RoutineToCall = MyDoEdit; break;
            case kSetVisibility:
                                RoutineToCall = MySetVisibility; break;
        }
    }
}

```

Color Picker Manager

```

case kDrawPicker:           RoutineToCall = MyDrawPicker; break;
case kItemHit:              RoutineToCall = MyItemHit; break;
case kSetBaseItem:          RoutineToCall = MySetBaseItem; break;
case kGetProfile:           RoutineToCall = MyGetProfile; break;
case kSetProfile:           RoutineToCall = MySetProfile; break;
case kGetPrompt:            RoutineToCall = MyGetPrompt; break;
case kSetPrompt:            RoutineToCall = MySetPrompt; break;
case kGetIconData:          RoutineToCall = MyGetIconData; break;
case kGetEditMenuState:     RoutineToCall = MyGetEditMenuState; break;
case kSetOrigin:            RoutineToCall = MySetOrigin; break;
case kExtractHelpItem:      RoutineToCall = MyExtractHelpItem; break;
default:                    return 0;
}
}
result=CallComponentFunctionWithStorage(storage, params, RoutineToCall);
return result;
}

```

“Color Picker–Defined Functions” in the chapter “Color Picker Manager Reference” in *Advanced Color Imaging Reference* on the enclosed CD describes the interfaces your color picker must provide to respond to these request codes. The next several sections provide examples of how your color picker can respond to most of these request codes.

Initializing Your Color Picker

The Color Picker Manager sends the `kInitPicker` request code after your color picker has set up all of its external data. (If the Color Picker Manager opens a color picker only to obtain a list of color pickers for the More Choices list in a dialog box, your color picker will not receive this message unless the user actually chooses the color picker.)

Your color picker code should use the `CallComponentFunctionWithStorage` function, which invokes a specified function of your color picker with the parameters originally provided by the Color Picker Manager. Your color picker passes these parameters by specifying the same component parameters structure that was received by your color picker’s main entry point. The `CallComponentFunctionWithStorage` function also provides a handle to the memory associated with the current connection. Your color picker uses this memory to store private data that it initializes in response to the `kInitPicker` request code.

Color Picker Manager

To access resources that are stored in your color picker, you can use the Component Manager functions `OpenComponentResFile` and `CloseComponentResFile`. Listing 2-17 illustrates how a color picker uses these functions to gain access to its private data, which it initializes in response to the `kInitPicker` request code.

Listing 2-17 Initializing private data for a color picker

```
pascal ComponentResult MyInitPicker(PickerStorageHndl storage,
                                   PickerInitData *data)
{
    GrafPtr      thePort;
    OSErr        error = noErr;
    PMColorPtr    theColor;
    RGBColor      rgb;
    short         resFile;

    /* open resource file */
    resFile = OpenComponentResFile((Component) (*storage)->myself);
    GetPort(&thePort);
    (*storage)->port = thePort;
    (*storage)->flags = data->flags;
    (*storage)->realPicker = true;
    /* always open an invisible picker */
    (*storage)->visible = false;
    (*storage)->active = true;
    /* initialize internal colors */
    theColor = &(*storage)->color;
    theColor->profile = 0;
    theColor->color.rgb.red = 0;
    theColor->color.rgb.green = 0;
    theColor->color.rgb.blue = 0;
    (*storage)->origColor = (*storage)->color;
    (*storage)->lastRGB = (*storage)->color.color.rgb;
    MyInitNumerics(storage);
    /* allocate patterns for the color rectangles */
    (*storage)->newColorPat = NewPixPat();
    (*storage)->origColorPat = NewPixPat();
    /* initialize rectangles to the correct colors */
    rgb = (*storage)->color.color.rgb;
```


Color Picker Manager

```

        MakeRGBPat((*storage)->newColorPat,&rgb);
        rgb = (*storage)->origColor.color.rgb;
        MakeRGBPat((*storage)->origColorPat,&rgb);
        (*storage)->profile = 0L;
        finish:
        CloseComponentResFile(resFile);
        return error;
    }

```

▲ **WARNING**

Do not leave any resource files open when your color picker is closed. Their maps will be left in the subheap when the subheap is freed, causing the Resource Manager to crash. ▲

Before handling the `kInitPicker` request code, your color picker must be able to handle the `kTestGraphicsWorld`, `kGetDialog`, and `kGetItemList` request codes.

In response to the `kTestGraphicsWorld` request code, you should test whether your color picker can operate under existing conditions and return `noErr` if it can, as illustrated in Listing 2-18.

Listing 2-18 Testing whether an environment can support your color picker

```

pascal ComponentResult MyTestGraphicsWorld(PickerStorageHndl storage,
                                           PickerInitData *data)
{
    #pragma unused(storage,data)
    OSErr err = noErr;
    long gLong;
    /* color picker requires Color QuickDraw */
    Gestalt(gestaltQuickdrawVersion,&gLong);
    return gLong >= gestalt8BitQD ? noErr : pickerCantLive;
}

```

Color Picker Manager

If your color picker uses its own dialog box, it should return a pointer to this dialog box in response to the `kGetDialog` request code. If your color picker uses the default dialog box, it should return `nil`, as illustrated here.

```
pascal ComponentResult MyGetDialog(PickerStorageHndl storage)
{
    #pragma unused (storage)
    return 0L;
}
```

In response to the `kGetItemList` request code, your color picker should return its dialog box items, as illustrated in Listing 2-19. The Color Picker Manager adds these items to the color picker dialog box.

Listing 2-19 Returning the dialog box items for a color picker

```
pascal long MyGetItemList(PickerStorageHndl storage)
{
    #pragma unused (storage)
    Handle theItems;
    short resFile;

    /* open resource file to get DITL */
    resFile = OpenComponentResFile((Component) (*storage)->myself);
    /* get the DITL and detach it so it won't go away after
       closing the file */
    theItems = GetResource ('DITL', kPickerDITL);
    if(theItems)
        DetachResource(theItems);
    CloseComponentResFile(resFile);          /* close the file */
    return (long) theItems;
}
```

A color picker must respond to the `kSetVisibility` request code by making itself either visible or invisible, as requested by the Color Picker Manager.

Handling Events for Your Color Picker

The Color Picker Manager sends the `kDrawPicker` request in response to an update event. Your color picker responds to this code by redrawing your color picker, as illustrated in Listing 2-20.

Listing 2-20 Redrawing a color picker

```
pascal ComponentResult MyDrawPicker(PickerStorageHndl storage)
{
    if((*storage)->visible)
    {
        MyDrawColorList(storage);
        MyDrawColorEditor(storage,true);
    }
    return noErr;
}
```

The Color Picker Manager calls the Event Manager function `BeginUpdate` before sending the `kDrawPicker` request code and the Event Manager function `EndUpdate` after sending the `kDrawPicker` request code.

The Color Picker Manager sends the `kEvent` request code so that your color picker can handle events that the Dialog Manager does not handle. A color picker responds to the `kEvent` request code by performing any event processing in addition to or instead of that normally performed by the Dialog Manager. Listing 2-21 illustrates how a color picker can perform such event processing.

Listing 2-21 Responding to events before handing them to the Dialog Manager

```
pascal ComponentResult MyDoEvent(PickerStorageHndl storage,
                                eventData *data)
{
    OSErr err = noErr;
    /* initialize so events are not filtered */
    data->handled = false;
    data->action = kDidNothing;
    if(data->event)
    {
```

Color Picker Manager

```

switch(data->event->what)
{
    case nullEvent:
        DoIdle(storage,data);
        break;
    case mouseDown:
        err = DoMouseDown(storage,data);
        break;
    case keyDown:
    case autoKey:
        err = DoKeyDown(storage,data);
        break;
    case keyUp:
        err = DoKeyUp(storage,data);
        break;
    case activateEvt:
        if(data->event->modifiers & activeFlag)
            ActivatePicker(storage);
        else
            DeactivatePicker(storage);
        break;
}
}
return err;
}

```

The Color Picker Manager sends the `kItemHit` request code to inform your color picker of an event in one of its items. In turn, your color picker responds to the event for the item reported in the `itemHit` field of an `ItemHitData` record, which is described in the chapter “Color Picker Manager Reference” in *Advanced Color Imaging Reference* on the enclosed CD. Listing 2-22 illustrates how a color picker can perform such event processing.

Listing 2-22 Responding to events in color picker items

```

pascal ComponentResult MyItemHit(PickerStorageHndl storage,
                                ItemHitData *data)
{
    #pragma unused(iMod)
    Handle theItem;

```

Color Picker Manager

```

short iType;
Rect iBox;

OSErr err = noErr;
data->action = kDidNothing;
GetDialogItem((*storage)->port,
              (*storage)->baseItem + data->itemHit, &iType,
              &theItem, &iBox);
switch(data->itemHit)
{
    case iRedText:
    case iGreenText:
    case iBlueText:
        /* don't update everything as the user types each key;
           update only after the user leaves an edit field */
        if(data->iMod != kKeyDown && data->itemHit != kMouseDown)
            CheckCurrentWorld(storage,data->itemHit);
        break;
    case iOrigColor:
        err = DoListClick(storage,data);
        break;
    case iNewColor:
        break;
}
return err;
}

```

The Color Picker Manager sends the `kEdit` request code to inform your color picker that the user has chosen one of the edit commands from the Edit menu (or typed its keyboard equivalent). If your color picker, rather than the Dialog Manager, needs to perform the editing command, your color picker should do so in response to this request code. For example, because the Dialog Manager does not perform the Undo command, your `MyDoEdit` function can instead. The editing command is passed to your function in the field `theEdit` of the `EditData` record pointed to in the `data` parameter. Listing 2-23 illustrates how a color picker can respond to this result code.

Listing 2-23 Handling events in the color picker's Edit menu

```

pascal ComponentResult MyDoEdit(PickerStorageHndl storage,
                                EditData *data)
{
    RGBColor rgb;

    switch(data->theEdit)
    {
        default:
            /* default behavior is appropriate */
            data->action = kDidNothing;
            data->handled = false;
            break;
        case kUndo:
            rgb = (*storage)->lastRGB;
            (*storage)->lastRGB = (*storage)->color.color.rgb;
            (*storage)->color.color.rgb = rgb;
            /* update the other internal data and then redraw */
            MySetSelectionColor(storage);
            MyUpdateColorText(storage);
            MyDrawColorRects(storage, false);
            data->action = kColorChanged;
            data->handled = true;
            break;
    }
    return noErr;
}

```

Returning and Setting Color Picker Information

The Color Picker Manager sends the `kGetColor` code to request your color picker to supply an original or a new color. In turn, your color picker returns either the original color or the new color, as shown in Listing 2-24.

Listing 2-24 Returning the original or the new color

```

pascal ComponentResult MyGetColor(PickerStorageHndl storage,
                                   ColorType whichColor,
                                   PMColorPtr color)
{
    /* copy the color here because the profile is always set as
       we want it to be (nil - the system profile/RGB space) */
    if(whichColor == kNewColor)
        *color = (*storage)->color;
    else
        *color = (*storage)->origColor;
    return noErr;
}

```

Similarly, the Color Picker Manager sends the `kSetColor` code to request your color picker to set an original or a new color. Listing 2-25 shows how a color picker can set these colors.

Listing 2-25 Setting colors

```

pascal ComponentResult MySetColor(PickerStorageHndl storage,
                                   ColorType whichColor, PMColorPtr color)
{
    Boolean    updateEditor = false;
    Boolean    textInvalid = false;
    CWorld     cworld;
    PMColor    myColor;
    long       csLong;

    OSErr      err = noErr;
    myColor = *color;          /* get your own copy */
    /* check whether a profile was included; if so, convert the color to
       system space */
    if(color->profile)
    {
        /* first ensure that ColorSync is available */
        if(Gestalt(gestaltColorMatchingVersion,&csLong) != noErr)
        {

```

Color Picker Manager

```

    err = colorSyncNotInstalled;
    goto fail;
}
/* create the color world and convert the color */
if(CWNewColorWorld(&cworld,myColor.profile,0L) == noErr)
{
    if(CWMatchColors(cworld,&myColor.color,1) != noErr)
    {
        err = badProfileError;
        CWDisposeColorWorld(cworld);
        goto fail;
    }
    CWDisposeColorWorld(cworld);
}
else
{
    err = badProfileError;
    goto fail;
}
}
myColor.profile = 0L;          /* it's in the system space now */
if(whichColor == kNewColor)
{
    (*storage)->color = *color;
    (*storage)->lastRGB = color->color.rgb;
    updateEditor = true;
}
else
{
    (*storage)->origColor = *color;
    /* make a new pattern for the original color */
    MakeRGBPat((*storage)->origColorPat,&color->color.rgb);
    if((*storage)->visible)
        DrawMYColorRects(storage,true);/* redraw original color rect */
}
if(updateEditor) {
    /* make some calls to update data structures and redraw the parts of
       the picker that need to be redrawn */
    SetSelectionColor(storage);
    UpdateColorText(storage);
    if((*storage)->visible)
    {

```


Color Picker Manager

```

        /* you don't call DoPickerDraw here because you don't need to
           redraw everything */
        DrawMYColorRects(storage,false);
        DrawMYColorEditor(storage,false);
    }
}
fail: C
    return err;
}

```

Listing 2-26 illustrates a color picker–defined function that responds to the `kGetIconData` request code. The color picker uses a `PickerIconData` structure (described in the chapter “Color Picker Manager Reference” in *Advanced Color Imaging Reference* on the enclosed CD) to return the script code and the resource ID of the icon family by which the color picker identifies itself in the More Choices list of color picker dialog boxes. (This list is shown in Figure 2-2 on page 2-5.)

Listing 2-26 Returning icon data

```

pascal ComponentResult MyGetIconData(PickerStorageHndl storage,
                                     PickerIconData *data)
{
    short          fref;
    OSErr          err = noErr;
    PickerIconData **mypdat;

    data->scriptCode = 0;
    data->iconSuiteID = kPickerData;
    fref = OpenComponentResFile((Component) (*storage)->myself);
    if(fref)
    {
        mypdat = (PickerIconData **)GetResource(kPickerDataType,
                                                kPickerData);

        if(mypdat)
            *data = **mypdat;
        else
            goto fail;
    }
    else

```

Color Picker Manager

```

        goto fail;
    CloseComponentResFile(fref);
    return err;
fail:
    DebugStr("\pProblem getting resource");
    return pickerResourceError;
}

```

The Color Picker Manager sends the `kGetPrompt` request code to obtain the prompt string currently used by your color picker. Your color picker should respond to this code by returning its string, as illustrated in Listing 2-27.

Listing 2-27 Returning the color picker's prompt

```

pascal ComponentResult MyGetPrompt(PickerStorageHndl storage,
                                   Str255 prompt)
{
    Handle    theItem;
    short     iType;
    Rect      iBox;

    GetDialogItem((*storage)->port, (*storage)->baseItem +
                  iPrompt, &iType, &theItem, &iBox);
    GetDialogItemText(theItem, prompt);
    return noErr;
}

```

Listing 2-28 illustrates a color picker–defined function that responds to the `kSetPrompt` request code by setting the prompt string used by the color picker.

Listing 2-28 Setting the color picker's prompt

```

pascal ComponentResult MySetPrompt(PickerStorageHndl storage,
                                   Str255 prompt)
{
    Handle theItem;
    short iType;
    Rect iBox;
}

```

Color Picker Manager

```

        GetDlgItem((*storage)->port, (*storage)->baseItem +
                    iPrompt, &iType, &theItem, &iBox);
        SetDlgItemText(theItem, prompt);
        return noErr;
    }

```

The Color Picker Manager sends the `kGetProfile` request code to obtain the destination color-matching profile currently used by your color picker. Your color picker should respond to this code by returning the destination profile, as illustrated in Listing 2-29.

Listing 2-29 Returning the destination profile

```

pascal ComponentResult MyGetProfile(PickerStorageHndl storage)
{
    Handle h;
    h = (Handle) (*storage)->profile;
    if(h)
        HandToHand(&h);
    return (long) h;
}

```

Listing 2-30 illustrates a color picker–defined function that responds to the `kSetProfile` request code by setting a new destination profile for use by the color picker.

Listing 2-30 Setting the destination profile

```

pascal ComponentResult MySetProfile(PickerStorageHndl storage,
                                     CMProfileHandle profile)
{
    CMProfileHandle    myProfile;

    OSErr err = noErr;
    /* Make a private copy of the profile, even though this
       picker doesn't do anything with profiles. This is necessary
       because the Color Picker Manager relies on color pickers to

```

Color Picker Manager

```

        store this data so that it doesn't have to duplicate the
        storage and waste memory. */
    if(myProfile = profile)
    {
        HandToHand((Handle *) &myProfile);
        if((err = MemError()) != noErr)
            goto fail;
    }
    (*storage)->profile = myProfile;
fail:
    return err;
}

```

The Color Picker Manager sends the `kGetEditMenuState` request code to obtain information about the desired state of the edit menu for your color picker. As illustrated in Listing 2-31, your color picker should respond to this code by returning edit menu information in a `MenuState` structure, which is described in detail in the chapter “Color Picker Manager Reference” in *Advanced Color Imaging Reference* on the enclosed CD.

Listing 2-31 Specifying how the Edit menu should be set

```

pascal ComponentResult MyGetEditMenuState(PickerStorageHndl storage,
                                           MenuState *mState)
{
    #pragma unused(storage)
    OSErr err = noErr;
    mState->cutEnabled = true;
    mState->copyEnabled = true;
    mState->pasteEnabled = true;
    mState->clearEnabled = true;
    mState->undoEnabled = true;
    strcpy(mState->undoString, "\pUndo");
    return err;
}

```

Summary of the Color Picker Manager

Constants and Data Types

```
enum { /* gestalt selector */
    gestaltColorPickerVersion = 'cpkr' /* returns version of Color Picker Manager */
};

typedef struct PMColor {
    CMProfileHandle    profile;    /* a handle to a profile */
    CMColor            color;      /* a color-matching structure */
} PMColor, *PMColorPtr;

typedef struct PrivatePickerRecord **picker;

/* actions returned to the application from DoPickerEvent */
enum PickerAction {
    kDidNothing,          /* no action worth reporting */
    kColorChanged,        /* user chose different color */
    kOkHit,               /* user clicked OK */
    kCancelHit,           /* user clicked Cancel */
    kNewPickerChosen,      /* user chose new color picker */
    kApplItemHit          /* Dialog Manager returned an item in an
                           application-owned dialog box */
};

typedef short PickerAction;

/* types of colors a picker must maintain */
enum ColorTypes {
    kOriginalColor,       /* the starting color--the one to change */
    kNewColor             /* the last color selected by the user */
};

typedef short ColorType;

/* types of edit operations that are sent with the kEdit message */
enum EditOperations {
    kCut,                 /* perform the Cut command */
    kCopy,                /* perform the Copy command */
};
```

CHAPTER 2

Color Picker Manager

```
kPaste,      /* perform the Paste command */
kClear,      /* perform the Clear command */
kUndo        /* perform the Undo command */
};

typedef short EditOperation;

/* Item hit modifiers. These are sent along with the itemHit message and
   inform the picker of what it was that caused the item hit. */
enum ItemHitModifiers {
    kMouseDown,      /* mouse-down event on item */
    kKeyDown,        /* key-down event in current edit item */
    kFieldEntered,    /* tab into an edit field */
    kFieldLeft,       /* tab out of an edit field */
    kCutOp,           /* cut in current edit field */
    kCopyOp,          /* copy in current edit field */
    kPasteOp,         /* paste in current edit field */
    kClearOp,         /* clear in current edit field */
    kUndoOp           /* undo in current edit field */
};

typedef short ItemModifier;

/* The dialog placement specifiers. These tell the picker manager where to
   place the picker dialog (used for system dialogs). */
enum DialogPlacementSpecifiers {
    kAtSpecifiedOrigin, /* place the top-left corner of the dialog box at
                           the point specified in the dialogOrigin field of
                           the color picker parameter block */
    kDeepestColorScreen, /* center the dialog box on the screen with the
                           greatest color depth */
    kCenterOnMainScreen /* center the dialog box on the main screen */
};

typedef short DialogPlacementSpec;

/* these flags may be set by the app and are passed through to the picker */
#define DialogIsMoveable      1 /* the user can move the dialog box */
#define DialogIsModal         2 /* the dialog box is modal */
#define CanModifyPalette      4 /* the picker is allowed to install a palette */
#define CanAnimatePalette     8 /* the picker is allowed to animate the palette */
#define AppIsColorSyncAware   16 /* The application is ColorSync aware and can
                                   therefore convert colors between spaces (that
                                   is, it can accept non-RGB colors) */
```

Color Picker Manager

```

/* these flags are set by the Color Picker Manager (overriding any
   application settings) */
#define InSystemDialog          32 /* the color picker is in a system-owned
                                   dialog box */
#define InApplicationDialog     64 /* the color picker is in an application-
                                   owned dialog box */
#define InPickerDialog         128 /* the color picker is in its own dialog box */
#define DetachedFromChoices     256 /* the color picker has been detached
                                   from the More Choices list */

/* Color Picker attributes (bits 23 to 0 in the componentFlags field of the
   component 'thng') */
#define CanDoColor              1 /* the color picker supports Color QuickDraw */
#define CanDoBlackWhite         2 /* the color picker supports Basic QuickDraw */
#define AlwaysModifiesPalette    4 /* the color picker will modify palette
                                   entries on indexed devices */

#define MayModifyPalette         8 /* the color picker will modify palette if
                                   told it can */
#define PickerIsColorSyncAware 16 /* the color picker is ColorSync aware and can
                                   accept non-RGB colors */
#define CanDoSystemDialog       32 /* the color picker supports a system-owned
                                   dialog box */
#define CanDoAppDialog          64 /* the color picker supports an application-
                                   owned dialog box */
#define HasOwnDialog            128 /* the color picker has its own dialog box */
#define CanDetach               256 /* the picker can detach from a system-owned
                                   dialog box */

typedef struct PickerInitData {
    short    scriptCode;      /* script code */
    short    iconSuiteID;     /* resource ID for icon family */
    ResType   helpResType;    /* resource type for help balloon */
    short    helpResID;       /* resource ID for help balloon */
} PickerInitData;

/* the application-defined event filter function for DoPickerEvent */
typedef pascal Boolean (*UserEventProc)(EventRecord *event);

/* the application-defined function for dynamically changing colors */
typedef pascal void (*ColorChangedProc)(long userData, PMColorPtr newColor);

```

CHAPTER 2

Color Picker Manager

```
enum EventForecasters {
    kNoForecast,           /* no forecast (e.g., an update event) */
    kMenuChoice,           /* this event causes a menu to be chosen */
    kDialogAccept,         /* the dialog box will be accepted */
    kDialogCancel,         /* the dialog box will be cancelled */
    kLeaveFocus,            /* the focus will leave the color picker */
    kPickerSwitch,         /* new color picker chosen in More Choices list */
    kNormalKeyDown,        /* a normal key down to an edit field */
    kNormalMouseDown       /* a normal click in the color picker's focus */
};

typedef short EventForecaster;

#define PickerComponentType 'cpkr'

typedef enum {
    kInitPicker,           /* request codes handled by a color picker */
    kTestGraphicsWorld,    /* initialize any private data */
    kTestOperability,      /* test operability on current system */
    kGetDialog,            /* if using own dialog box, return a pointer to the dialog
                           box; if using the default dialog box, return nil */
    kGetItemList,          /* return a list of items for dialog box */
    kGetColor,             /* return original or last chosen color */
    kSetColor,             /* change original or last chosen color */
    kEvent,               /* perform any special processing necessary for an event */
    kEdit,                /* perform an editing command */
    kSetVisibility,        /* make color picker visible or invisible */
    kDrawPicker,           /* redraw color picker */
    kItemHit,             /* respond to event in a dialog box item */
    kSetBaseItem,          /* set base item for dialog box items */
    kGetProfile,           /* return a handle to the destination profile */
    kSetProfile,           /* change the destination profile */
    kGetPrompt,           /* return prompt string */
    kSetPrompt,           /* set a new prompt */
    kGetIconData,          /* return script code and resource ID of icon family */
    kGetEditMenuState,     /* return information about edit menu */
    kSetOrigin,           /* update any information about local
                           coordinate system of dialog box */
    kExtractHelpItem       /* return information about help balloons */
} PickerMessages;

typedef struct MenuItemInfo {
    short editMenuID;      /* resource ID of the edit menu */
    short cutItem;         /* item number of Cut command */
};
```


Color Picker Manager

```

    short copyItem;          /* item number of Copy command */
    short pasteItem;         /* item number of Paste command */
    short clearItem;         /* item number of Clear command */
    short undoItem;          /* item number of Undo command */
} MenuItemInfo;

typedef struct MenuState {
    Boolean    cutEnabled;    /* whether Cut menu item is enabled */
    Boolean    copyEnabled;   /* whether Copy menu item is enabled */
    Boolean    pasteEnabled;  /* whether Paste menu item's enabled */
    Boolean    clearEnabled;  /* whether Clear menu item's enabled */
    Boolean    undoEnabled;   /* whether Undo menu item is enabled */
    Str255     undoString;    /* text for Undo menu item */
} MenuState;

typedef struct ColorPickerInfo {          /* color picker parameter block */
    PColor          theColor;             /* a picker color */
    CMProfileHandle dstProfile;            /* profile for destination device */
    long            flags;                 /* color picker flags */
    DialogPlacementSpec placeWhere;        /* dialog box placement specifier */
    Point           dialogOrigin;          /* upper-left corner of dialog box */
    long            pickerType;            /* color picker type */
    UserEventProc   eventProc;             /* event filter function */
    ColorChangedProc colorProc;            /* color change function */
    long            colorProcData;         /* data for color change function */
    Str255          prompt;                /* color picker prompt */
    MenuItemInfo    mInfo;                 /* application's edit menu items */
    Boolean          newColorChosen;       /* whether user changed color */
} ColorPickerInfo;

typedef struct SystemDialogInfo {
    long            flags;                 /* color picker flags */
    long            pickerType;            /* color picker type */
    DialogPlacementSpec placeWhere;        /* dialog box placement specifier */
    Point           dialogOrigin;          /* upper-left corner of dialog box */
    MenuItemInfo    mInfo;                 /* application's Edit menu items */
} SystemDialogInfo;

typedef struct PickerDialogInfo {
    long            flags;                 /* color picker flags */
    long            pickerType;            /* color picker type */

```

CHAPTER 2

Color Picker Manager

```
    Point          *dialogOrigin;    /* upper-left corner of dialog box */
    MenuItemInfo    mInfo;            /* application's menu items */
} PickerDialogInfo;

typedef struct ApplicationDialogInfo {
    long            flags;            /* color picker flags */
    long            pickerType;       /* color picker type */
    DialogPtr       theDialog;        /* pointer to dialog box */
    Point           pickerOrigin;     /* upper-left corner of dialog box */
    MenuItemInfo    mInfo;            /* application's Edit menu items */
} ApplicationDialogInfo;

typedef struct EventData {
    EventRecord      *event;          /* an event record */
    PickerAction     action;          /* action performed by color picker */
    short            itemHit;         /* the item number for the item
                                     associated with the event */

    Boolean          handled;         /* true if color picker handled event */
    ColorChangedProc colorProc;       /* application-defined function for
                                     changing colors in the document */

    long             colorProcData;   /* data used by application for function
                                     in ColorChangedProc field */

    EventForecaster  forecast;        /* event forecaster */
} EventData;

typedef struct EditData {
    EditOperation    theEdit;         /* the editing operation */
    PickerAction     action;          /* action performed by color picker */
    Boolean          handled;         /* whether action was handled */
} EditData;

typedef struct ItemHitData {
    short            itemHit;         /* item receiving event */
    ItemModifier     iMod;            /* type of event */
    PickerAction     action;          /* picker's action */
    ColorChangedProc colorProc;       /* color-changed function */
    long             colorProcData;   /* data for color-changed function */
    Point            where;           /* mouse location */
} ItemHitData;

typedef struct HelpItemInfo {
    long             options;         /* 'hmnu' options bits */
    Point            tip;             /* tip location */
}
```

Color Picker Manager

```

    Rect                altRect;          /* alternate rectangle */
    short               theProc;          /* res ID of balloon-definition function */
    short               variant;          /* variation code */
    HMMessageRecord     helpMessage;      /* help message structure */
} HelpItemInfo;

typedef unsigned short SmallFract;        /* unsigned fraction between 0 and 1 */
enum {MaxSmallFract = 0x0000FFFF};       /* Maximum small fract value, as long */

struct HSVColor {
    SmallFract hue;                /* fraction of circle, red at 0 */
    SmallFract saturation;         /* 0-1, 0 for gray, 1 for pure color */
    SmallFract value;             /* 0-1, 0 for black, 1 for maximum intensity */
};
typedef struct HSVColor HSVColor;

struct HSLColor {
    SmallFract hue;                /* fraction of circle, red at 0 */
    SmallFract saturation;         /* 0-1, 0 for gray, 1 for pure color */
    SmallFract lightness;         /* 0-1, 0 for black, 1 for white */
};
typedef struct HSLColor HSLColor;

struct CMYColor {
    SmallFract cyan;              /* cyan component */
    SmallFract magenta;           /* magenta component */
    SmallFract yellow;            /* yellow component */
};
typedef struct CMYColor CMYColor;

```

Color Picker Manager Functions

Using the Standard Color Picker Dialog Box

```

pascal OSErr PickColor              (ColorPickerInfo *theColorInfo);

pascal Boolean GetColor              (Point where,
                                     Str255 prompt,
                                     RGBColor *inColor,
                                     RGBColor *outColor);

```

Creating a Customized Color Picker Dialog Box

```

pascal OSErr CreateColorDialog      (SystemDialogInfo *info,
                                     picker *thePicker);

pascal OSErr CreatePickerDialog     (PickerDialogInfo *info,
                                     picker *thePicker);

pascal OSErr AddPickerToDialog      (ApplicationDialogInfo *info,
                                     picker *thePicker);

pascal OSErr SetPickerVisibility    (picker thePicker,
                                     short visible);

pascal OSErr GetPickerVisibility    (picker thePicker,
                                     Boolean *visible);

pascal OSErr SetPickerPrompt        (picker thePicker,
                                     Str255 promptString);

pascal OSErr GetPickerOrigin        (picker thePicker,
                                     Point *where);

pascal OSErr SetPickerOrigin        (picker thePicker,
                                     Point where);

pascal OSErr DisposeColorPicker     (picker thePicker);

```

Handling Events in a Color Picker Dialog Box

```

pascal OSErr DoPickerEvent          (picker thePicker,
                                     EventData *data);

pascal OSErr DoPickerEdit           (picker thePicker,
                                     EditData *data);

pascal OSErr DoPickerDraw           (picker thePicker);

```

Getting Colors From and Setting Colors for a Color Picker

```

pascal OSErr GetPickerColor         (picker thePicker,
                                     ColorType whichColor,
                                     PMColor *color);

pascal OSErr SetPickerColor         (picker thePicker,
                                     ColorType whichColor,
                                     PMColor *color);

```

Getting the Menu State and the Help Balloons for a Color Picker

```
pascal OSErr GetPickerEditMenuState (
                                picker thePicker,
                                MenuState *mState);

pascal OSErr ExtractPickerHelpItem (
                                picker thePicker,
                                short itemNo,
                                short whichState,
                                HelpItemInfo *helpInfo);
```

Getting and Setting Color-Matching Profiles for a Color Picker

```
pascal OSErr GetPickerProfile    (picker thePicker,
                                CMPProfileHandle *profile);

pascal OSErr SetPickerProfile    (picker thePicker,
                                CMPProfileHandle profile);
```

Converting Colors Among Color Models

```
pascal void CMY2RGB              (const CMYColor *cColor,
                                RGBColor *rColor);

pascal void RGB2CMY              (const RGBColor *rColor,
                                CMYColor *cColor);

pascal void HSL2RGB              (const HSLColor *hColor,
                                RGBColor *rColor);

pascal void RGB2HSL              (const RGBColor *rColor,
                                HSLColor *hColor);

pascal void HSV2RGB              (const HSVColor *hColor,
                                RGBColor *rColor);

pascal void RGB2HSV              (const RGBColor *rColor,
                                HSVColor *hColor);
```

Converting Between SmallFract and Fixed Values

```
pascal SmallFract Fix2SmallFract (Fixed f)

pascal Fixed SmallFract2Fix      (SmallFract s);
```

Application-Defined Functions

Handling Application-Directed Events in a Color Picker

```
pascal Boolean MyPickerFilterFunction (
                                EventRecord *event)
```

Changing Colors in a Document

```
pascal void MyColorChangedFunction (
                                long userData,
                                PMLColorPtr newColor)
```

Color Picker–Defined Functions

Responding to Creation and Initialization Requests

```
pascal long MyTestGraphicsWorld (PickerStorageHndl storage,
                                PickerInitData *data);

pascal long MyInitPicker (PickerStorageHndl storage,
                        PickerInitData *data);

pascal DialogPtr MyGetDialog (PickerStorageHndl storage);

pascal long MyDrawPicker (PickerStorageHndl storage);

pascal long MySetVisibility (PickerStorageHndl storage,
                        Boolean visible);
```

Responding to Requests to Return and Set Color Picker Information

```
pascal long MyGetColor (PickerStorageHndl storage,
                        ColorType whichColor,
                        PMLColorPtr color);

pascal long MySetColor (PickerStorageHndl storage,
                        ColorType whichColor,
                        PMLColorPtr color);

pascal long MyGetItemList (PickerStorageHndl storage);

pascal long MySetBaseItem (PickerStorageHndl storage,
                        short baseItem);
```

```

pascal long MyGetIconData      (PickerStorageHndl storage,
                                PickerIconData *data);

pascal long MyGetPrompt        (PickerStorageHndl storage,
                                Str255 prompt);

pascal long MySetPrompt        (PickerStorageHndl storage,
                                Str255 prompt);

pascal long MySetOrigin        (PickerStorageHndl storage,
                                Point where);

pascal CMPProfileHandle MyGetProfile (
                                PickerStorageHndl storage);

pascal long MySetProfile        (PickerStorageHndl storage,
                                CMPProfileHandle profile);

pascal long MyGetEditMenuState (PickerStorageHndl storage,
                                MenuState *mState);

pascal long MyExtractHelpItem  (PickerStorageHndl storage,
                                short itemNo,
                                short whichMsg,
                                HelpItemInfo *helpInfo);

```

Responding to Events in a Color Picker

```

pascal long MyDoEvent          (PickerStorageHndl storage,
                                EventData *data);

pascal long MyItemHit          (PickerStorageHndl storage,
                                ItemHitData *data);

pascal long MyDoEdit           (PickerStorageHndl storage,
                                EditData *data);

```


Introduction to the ColorSync Manager

Contents

Introduction to Color and Color Management Systems	3-4
Color: A Brief Overview	3-4
Color Perception	3-5
Hue, Saturation, and Brightness	3-5
Additive and Subtractive Color	3-6
Color Spaces	3-6
Gray Spaces	3-7
RGB-Based Color Spaces	3-7
CMY-Based Color Spaces	3-10
Device-Independent Color Spaces	3-11
Indexed Color Spaces	3-14
Color-Component Values, Color Values, and Colors	3-15
Color Conversion and Color Matching	3-15
Color Management Systems	3-17
About the ColorSync Manager	3-18
Programming Interfaces	3-18
About the ColorSync Manager's Memory Allocation and Use	3-19
Profiles	3-19
Color Management Modules	3-22
When Color Matching Occurs	3-24
QuickDraw GX and the ColorSync Manager	3-26
What Users Can Do With ColorSync-Supportive Applications	3-27
Display Matching	3-27
Gamut Checking	3-27
Soft Proofing	3-28
Device-Linked Profiles	3-28
Calibration	3-28

This chapter describes the **ColorSync Manager**, which provides your application or color peripheral device driver with device-independent color-matching and color conversion services. The ColorSync Manager allows you to match colors accurately across different input, display, and output devices. You can also convert colors easily and exactly across color spaces belonging to the same base family. Color spaces, color matching, and color conversion are discussed throughout this chapter.

Read this chapter if your software product performs color drawing, printing, or calculation or if your peripheral device supports color. You should also read this chapter if you are creating a color management module (CMM) to be used for color matching, color gamut checking, and profile management.

The ColorSync Manager provides your application, device driver, or CMM with color-matching capabilities that your users can employ without the need for a proprietary environment. The ColorSync Manager provides the first system-level implementation of an industry-standard color-matching system.

The ColorSync Manager is contained in a system extension. To provide its color-matching services, the ColorSync Manager uses one or more color management modules (CMMs) and profiles. A **color management module (CMM)** is a component that implements color-matching and gamut-checking services. Your application or driver can supply its own CMM, or you can use the robust default CMM that Apple supplies. A **profile** provides a means of defining the color characteristics of a given device in a particular state. A separate control panel, the ColorSync™ System Profile control panel, allows the user to specify the ColorSync system profile. The **system profile** defines the color characteristics for the system's display device. CMMs and profiles are discussed throughout this chapter.

The ColorSync Manager relies on the Component Manager for the basis of the framework that allows plug-and-play capability for third-party CMMs.

The next section, "Introduction to Color and Color Management Systems," provides a general introduction to color models and to approaches taken by the publishing and computer industries to manage color.

"About the ColorSync Manager," beginning on page 3-18, provides a general introduction to Apple Computer's implementation of the ColorSync color management system.

Introduction to Color and Color Management Systems

Different types of color peripheral devices (such as displays, scanners, and printers) use different methods for representing color information and are capable of producing different ranges of colors. As a result, colors do not match consistently when scanned or drawn to many different imaging devices. To address this problem, the ColorSync Manager allows users to transparently move color images from one device to another and from one operating system to another. But before learning about the ColorSync Manager, it is necessary to have a basic knowledge of color.

This chapter provides a general introduction to the basics of color and color management systems. Read this chapter to learn about color perception, additive and subtractive color systems, how different peripheral devices represent color, and how color management systems maintain consistent color among devices.

For more information on color theory and color spaces, you may also want to read other books such as these:

- Fred W. Billmeyer, Jr., and Max Saltzman. *Principles of Color Technology*, second edition. Wiley, 1981.
- James D. Foley. *Computer Graphics: Principles and Practice*, second edition. Addison-Wesley, 1990.
- Roy Hall. *Illumination and Color in Computer Generated Imagery*. New York: Springer-Verlag, 1988.
- R.W.G. Hunt. *Measuring Colour*, second edition. Prentice-Hall, 1991.

Color: A Brief Overview

Color is created through the interaction of a light, an object, and the eye. There must be a light to illuminate the object. White light contains many different colors of light. This can be seen by observing how sunlight is broken into its components when passed through a prism. The resulting rainbow represents the “visible spectrum” consisting of the colors that can be seen by the eye. Each color of light has a particular wavelength. An object appears to be a certain

color because it has pigments that absorb some of the wavelengths of the light that illuminates it while reflecting others back to the eye.

Color Perception

The eye contains three types of cone receptors. Each receptor is sensitive to about one-third of the visible spectrum. These are blue light, green light, and red light. The color the eyes see in an object depends on how much red, green, and blue light is reflected to the eye. Black is perceived when no light is reflected to the eye. When red, green, and blue lights are reflected to the eye in equal amounts, then white is perceived.

Even the conditions in which color is viewed greatly affect the perception of color. The light source and environment must be standardized for accurate viewing. When viewing colors, people in the graphic arts industry, for example, avoid fluorescent and tungsten lighting, use a particular illuminant, and proof against a neutral gray surface.

Color images frequently contain hundreds of distinctly different colors. To reproduce such images on a color peripheral device would be impractical. However, a very broad range of colors can be visually matched by a mixture of three “primary” lights. This allows colors to be reproduced on a display by a mixture of red, green, and blue lights or on a printer by a mixture of cyan, magenta, and yellow inks or pigments. Cyan absorbs red, magenta absorbs green, and yellow absorbs blue. Black is printed to increase contrast and make up for the deficiency of the inks.

The use of only three colors to reproduce thousands of colors is possible because the eyes are basically responsive to these three broad sections of the spectrum. The three color values constitute the specification for the matching of properties of a color.

Hue, Saturation, and Brightness

Color is described as having three dimensions. These dimensions are hue, saturation, and brightness. **Hue** is the name of the color, which places the color in its correct position in the spectrum. For example, if a color is described as blue, it is distinguished from yellow, red, green, or other colors. **Saturation** refers to the degree of hue in a color, or a color’s strength. A neutral gray is considered to have zero saturation. A saturated red would have a color similar to apple red. **Brightness** is the term used to describe differences in the intensity of light reflected from or transmitted by a color image. The hue of an object

may be blue, but the terms *dark* and *light* distinguish the brightness of one object from another.

Additive and Subtractive Color

The **additive color theory** refers to the process of mixing red, green, and blue lights, which are each approximately one-third of the visible spectrum. Additive color theory explains how red, green, and blue light can be added to make white light. Red and green projected together produce yellow, red and blue produce magenta, and blue and green produce cyan. With transmitted light, all the colors of the rainbow can be matched.

The **subtractive color theory** refers to the process of combining subtractive colorants such as inks or dyes. In this theory colorants of cyan, magenta, and yellow are used to subtract a portion of the white light that is illuminating an object. The color of an object is the result of the color lights that are not absorbed by the object. An apple appears that red because the surface of the apple absorbs the blue and green light.

Color Spaces

A **color space** is a model for representing color in terms of intensity values; a color space specifies how color information is represented. It defines a one-, two-, three-, or four-dimensional space whose dimensions, or **components**, represent intensity values. A color component is also referred to as a **color channel**. For example, RGB space is a three-dimensional color space whose components are the red, green, and blue intensities that make up a given color. Visually, these spaces are often represented by various solid shapes, such as cubes, cones, or polyhedra.

The ColorSync Manager directly supports several different color spaces to give you the convenience of working in whatever kind of color data most suits your needs. The ColorSync color spaces fall into several groups, or base families. They are

- **gray spaces**, used for grayscale display and printing
- **RGB-based color spaces**, used mainly for displays and scanners
- **CMYK-based color spaces**, used mainly for color printing

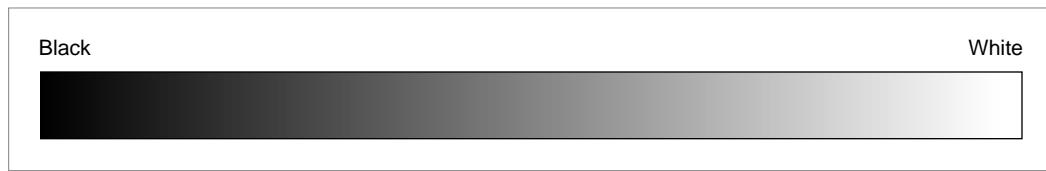
- **device-independent color spaces**, used mainly for color models
- **heterogeneous HiFi color spaces**, also referred to as multichannel color spaces, primarily used in new printing processes involving the use of gold plate and silver, and also for spot coloring

All color spaces within a base family differ only in details of storage format or else are related to each other by very simple mathematical formulas.

Gray Spaces

Gray spaces typically have a single component, ranging from black to white, as shown in Figure 3-1. Gray spaces are used for black-and-white and grayscale display and printing.

Figure 3-1 Gray space



RGB-Based Color Spaces

The **RGB space** is a three-dimensional color space whose components are the red, green, and blue intensities that make up a given color. For example, scanners read the amounts of red, green, and blue light that are reflected from an image and then convert those amounts into digital values. Displays receive the digital values and convert them into red, green, and blue light seen onscreen.

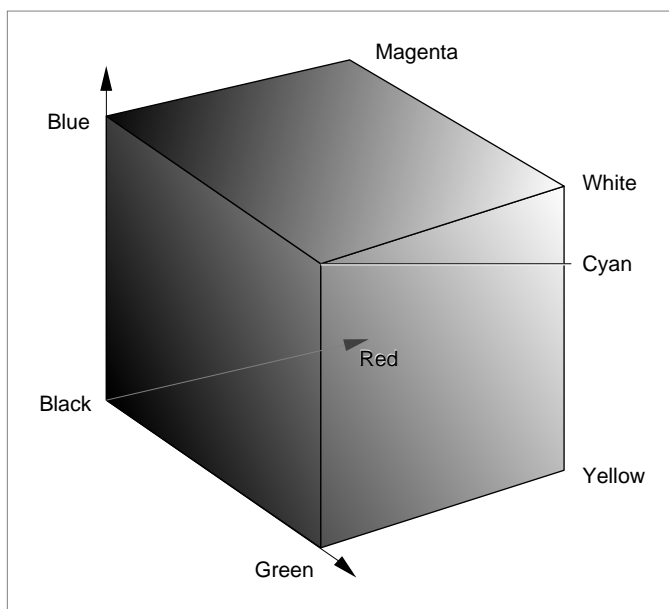
RGB-based color spaces are the most commonly used color spaces in computer graphics, primarily because they are directly supported by most color displays and scanners. RGB color spaces are device dependent and additive. The groups of color spaces within the RGB base family include

- RGB spaces
- HSV and HLS spaces

RGB Spaces

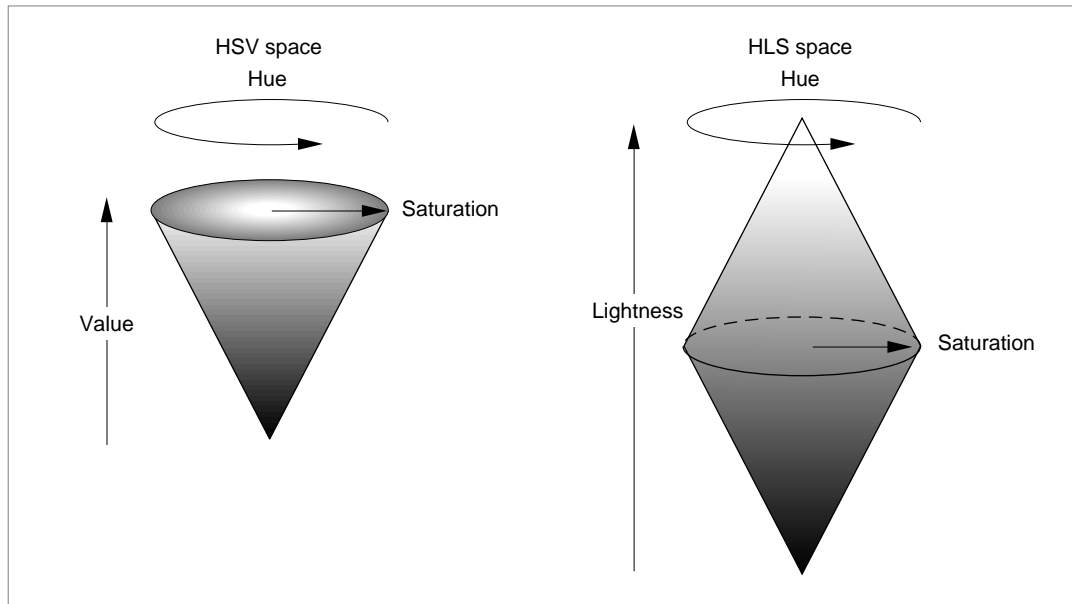
Any color expressed in RGB space is some mixture of three primary colors: red, green, and blue. Most RGB-based color spaces can be visualized as a cube, as in Figure 3-2, with corners of black, the three primaries (red, green, and blue), the three secondaries (cyan, magenta, and yellow), and white. (See also Color Plate 1 at the front of this book.)

Figure 3-2 RGB color space



HSV and HLS Color Spaces

HSV space and **HLS space** are transformations of RGB space that allow colors to be described in terms more natural to an artist. The name *HSV* stands for *hue*, *saturation*, and *value*, and *HLS* stands for *hue*, *lightness*, and *saturation*. The two spaces can be thought of as being single and double cones, as shown in Figure 3-3. (See also Color Plate 2 at the front of this book.)

Figure 3-3 HSV color space and HLS color space

The components in HLS space are analogous, but not completely identical, to the components in HSV space:

- The hue component in both color spaces is an angular measurement, analogous to position around a color wheel. A hue value of 0 indicates the color red; the color green is at a value corresponding to 120°, and the color blue is at a value corresponding to 240°. Horizontal planes through the cones in Figure 3-3 are hexagons; the primaries and secondaries (red, yellow, green, cyan, blue, and magenta) occur at the vertices of the hexagons.
- The saturation component in both color spaces describes color intensity. A saturation value of 0 (in the middle of a hexagon) means that the color is “colorless” (gray); a saturation value at the maximum (at the outer edge of a hexagon) means that the color is at maximum “colorfulness” for that hue angle and brightness.
- The value component (in HSV space) and the lightness component (in HLS space) describe brightness or luminance. In both color spaces, a value of 0 represents black. In HSV space, a maximum value means that the color is at

its brightest. In HLS space, a maximum value for lightness means that the color is white, regardless of the current values of the hue and saturation components. The brightest, most intense color in HLS space occurs at a lightness value of exactly half the maximum.

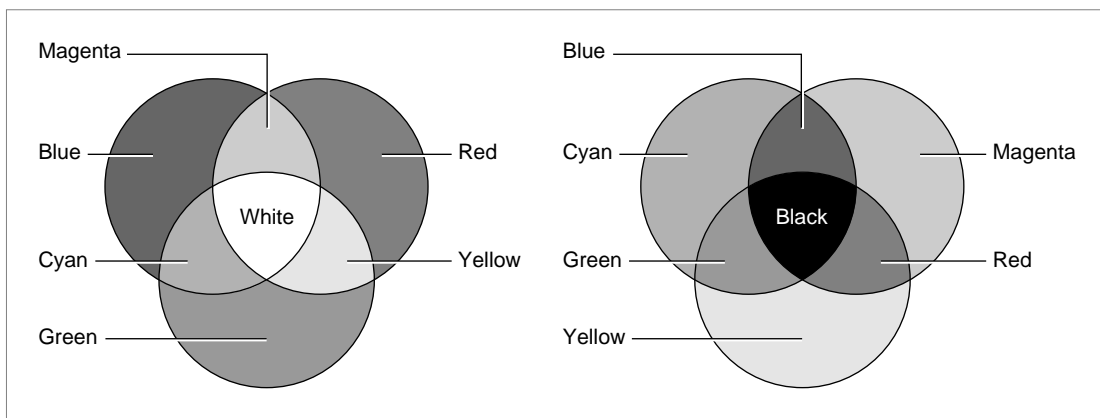
CMY-Based Color Spaces

CMY-based color spaces are most commonly used in color printing systems. They are device dependent and subtractive in nature. The groups of color spaces within the CMY family include

- **CMY**, which is not very common except on low-end color printers
- **CMYK**, which models the way inks or dyes are applied to paper in printing

The name *CMYK* refers to cyan, magenta, yellow, and black. Cyan, magenta, and yellow are the three primary colors in this color space, and red, green, and blue are the three secondaries. Theoretically black is not needed. However, when full-saturation cyan, magenta, and yellow inks are mixed equally on paper, the result is usually a dark brown, rather than black. Therefore, black ink is overprinted in darker areas to give a better appearance. Figure 3-4 shows how additive colors expressed in CMYK space and subtractive colors expressed in RGB space mix to form other colors. (See also Color Plates 1 and 3 at the front of this book.)

Figure 3-4 Additive colors expressed in CMYK and subtractive colors expressed in RGB



Theoretically, the relation between RGB values and CMY values in CMYK space is quite simple:

Cyan	=	1.0 - red
Magenta	=	1.0 - green
Yellow	=	1.0 - blue

(where red, green, and blue intensities are expressed as fractional values varying from 0 to 1). In reality, the process of deriving the cyan, magenta, yellow, and black values from a color expressed in RGB space is complex, involving device-specific, ink-specific, and even paper-specific calculations of the amount of black to add in dark areas (black generation) and the amount of other ink to remove (undercolor removal) where black is to be printed. The ColorSync Manager performs those calculations for you when converting among color spaces.

Device-Independent Color Spaces

Some color spaces allow color to be expressed in a device-independent way. Whereas RGB colors vary with display and scanner characteristics, and CMYK colors vary with printer, ink, and paper characteristics, *device-independent colors* are meant to be true representations of colors as perceived by the human eye. These color representations, called **device-independent color spaces**, result from work carried out in 1931 by the Commission Internationale d'Eclairage (CIE) and for that reason are also called **CIE-based color spaces**.

The most common method of identifying color within a color space is a three-dimensional geometry. The three color attributes, hue, saturation, and brightness, are measured, assigned numeric values, and plotted within the color space.

RGB colors vary with display characteristics, and CMYK colors vary with printer, ink, and paper characteristics. Conversion from an RGB color space to a CMYK color space involves a number of variables. The type of printer or printing press, the paper stock, and the inks used all influence the balance between cyan, magenta, yellow, and black. In addition, different devices have different **gamuts**, or ranges of colors that they can produce. Because the colors produced by RGB and CMYK specifications vary from device to device, they're called device-dependent color spaces. Device color spaces enable the specification of color values that are directly related to their representation on a particular device.

Device-independent color spaces, or **interchange color spaces**, are used for the interchange of color data from the native color space of one device to the native color space of another device.

The CIE created a set of color spaces that specify color in terms of human perception. It then developed algorithms to derive three imaginary primary constituents of color—X, Y, and Z—that can be combined at different levels to produce all the color the human eye can perceive. The resulting color model, CIE, and other CIE color models form the basis for all color management systems. Although the RGB and CMYK values differ from device to device, human perception of color remains consistent across devices. Colors can be specified in the CIE-based color spaces in a way that is independent of the characteristics of any particular display or reproduction device. The goal of this standard is for a given CIE-based color specification to produce consistent results on different devices, up to the limitations of each device.

CIELUV is a CIE-based color space used for representing additive color systems, including color lights and emissive phosphor displays. CIELAB is an independent color space used for representing subtractive systems, where light is absorbed by colorants such as inks and dyes. (See Color Plate 4 at the front of this book.)

XYZ Space

There are several CIE-based color spaces, but all are derived from the fundamental **XYZ space**. The XYZ space allows colors to be expressed as a mixture of the three **tristimulus values** X, Y, and Z. The term *tristimulus* comes from the fact that color perception results from the retina of the eye responding to three types of stimuli. After experimentation, the CIE set up a hypothetical set of primaries, XYZ, that correspond to the way the eye's retina behaves.

The CIE defined the primaries so that all visible light maps into a positive mixture of X, Y, and Z, and so that Y correlates approximately to the apparent lightness of a color. Generally, the mixtures of X, Y, and Z components used to describe a color are expressed as percentages ranging from 0 percent up to, in some cases, just over 100 percent.

Other device-independent color spaces based on XYZ space are used primarily to relate some particular aspect of color or some perceptual color difference to XYZ values.

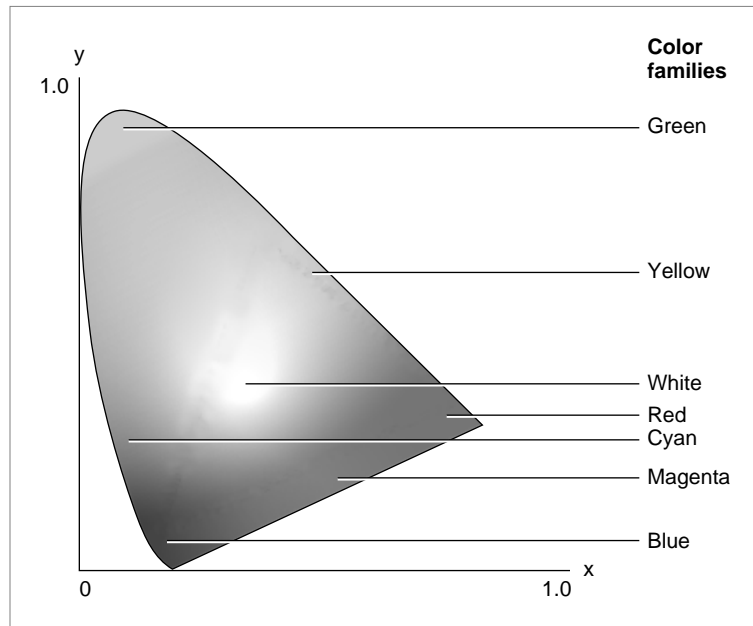
Yxy Space

Yxy space expresses the XYZ values in terms of x and y chromaticity coordinates, somewhat analogous to the hue and saturation coordinates of HSV space. The coordinates are shown in the following formulas, used to convert XYZ into Yxy:

$$\begin{aligned} Y &= Y \\ x &= X / (X+Y+Z) \\ y &= Y / (X+Y+Z) \end{aligned}$$

Note that the Z tristimulus value is incorporated into the new coordinates and does not appear by itself. Since Y still correlates to the lightness of a color, the other aspects of the color are found in the chromaticity coordinates x and y. This allows color variation in Yxy space to be plotted on a two-dimensional diagram. Figure 3-5 shows the layout of colors in the x and y plane of Yxy space.

Figure 3-5 Yxy chromaticities in the CIE color space



L*u*v* Space and L*a*b* Space

One problem with representing colors using the XYZ and Yxy color spaces is that they are perceptually nonlinear: it is not possible to accurately evaluate the perceptual closeness of colors based on their relative positions in XYZ or Yxy space. Colors that are close together in Yxy space may seem very different to observers, and colors that seem very similar to observers may be widely separated in Yxy space.

L*u*v* space is a nonlinear transformation of XYZ space in order to create a perceptually linear color space. **L*a*b* space** is a nonlinear transformation (that is, a third-order approximation) of the Munsell color-notation system (which is not described here). Both are designed to match perceived color difference with quantitative distance in color space.

Both L*u*v* space and L*a*b* space represent colors relative to a **reference white point**, which is a specific definition of what is considered white light, represented in terms of XYZ space, and usually based on the whitest light that can be generated by a given device. (In that sense L*u*v* and L*a*b* are not completely device independent; two numerically equal colors are truly identical only if they were measured relative to the same white point.)

Measuring colors in relation to a white point allows for color measurement under a variety of illuminations.

A primary benefit of using L*u*v* space and L*a*b* space is that the perceived difference between any two colors is proportional to the geometric distance in the color space between their color values. For applications where closeness of color needs to be quantified, such as in colorimetry, gemstone evaluation, or dye matching, use of L*u*v* space or L*a*b* space is common.

Indexed Color Spaces

In situations where you use only a limited number of colors, it can be impractical or impossible to specify colors directly. If you have a bitmap with only a few bits per pixel (1, 2, 4, or 8, for example), each pixel is too small to contain a complete color specification; its color must be specified as an index into a list or table of color values. If you are using spot colors in printing or pen colors in plotting, it can be simpler and more precise to specify each color as an index into a list of colors instead of an actual color value. Also, if you want to restrict the user to drawing with a specific set of colors, you can put the colors in a list and specify them by index.

Indexed space is the color space you use when drawing with indirectly specified colors. An indexed color value (a color specification in indexed color space) consists of an index value that refers to a color in a color list. Color values are defined in the next section.

Color-Component Values, Color Values, and Colors

Each of the color spaces described in this chapter requires one or more numeric values in a particular format to specify a color.

Each dimension, or component, in a color space has a **color-component value**. A color-component value can vary from 0 to 65,535 (0xFFFF), although the numerical interpretation of that range is different for different color spaces. In most cases, color-component intensities are interpreted numerically as varying between 0 and 1.0.

Depending on the color space used, one, two, three, or four color-component values combine to make a **color value**. For HiFi colors, up to eight color-component values combine to make a color. A color value is a structure; it is the complete specification of a color in a given color space.

Color Conversion and Color Matching

Color conversion is the process of converting colors from one color space to another. **Color matching**, which entails color conversion, is the process of adjusting or *matching* these converted colors appropriately to achieve maximum similarity from the gamut of one color space to the other. Color matching always involves color conversion, whereas color conversion may not entail color matching.

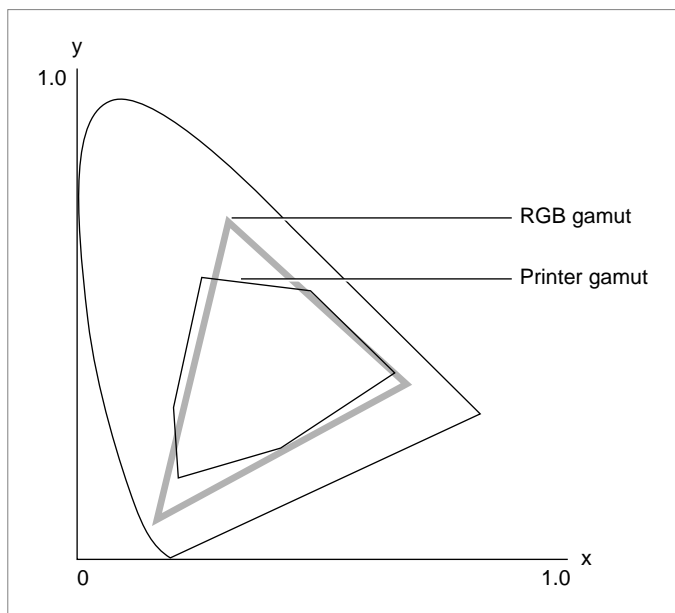
Different imaging devices (scanners, displays, printers) work in different color spaces, and each can have a different gamut. Color displays from different manufacturers all use RGB colors but may have different RGB gamuts. Printers that work in CMYK space vary drastically in their gamuts, especially if they use different printing technologies. Even a single printer's gamut can vary significantly with the ink or type of paper it uses. It's easy to see that conversion from RGB colors on an individual display to CMYK colors on an individual printer using a particular paper type can lead to unpredictable results.

When an image is output to a particular device, the device displays only those colors that are within its gamut. Likewise, when an image is created by

scanning, only those colors within the scanner's gamut are saved. Devices with different gamuts cannot reproduce each other's colors exactly, but careful shifting of the colors used on one device can improve the visual match when the image is displayed on another.

Figure 3-6 shows examples of two devices' color gamuts, projected onto Yxy space. Both devices produce less than the total possible range of colors, and the printer gamut is restricted to a significantly smaller range than the RGB gamut. The problem illustrated by Figure 3-6 is to be able to display the same image on both devices with a minimum of visual mismatch. The solution to the problem is to match the colors of the image using profiles for both devices and one or more color management modules (CMMs).

Figure 3-6 Color gamuts for two devices expressed in Yxy space



Color Management Systems

Members of the computer and publishing industries have developed color management systems (CMSs) to convert colors from the color space of one device to the color space of another device. The goal of these systems is to provide consistent color across peripheral devices and across operating-system platforms. Most CMSs are proprietary. The ColorSync Manager, however, supports an industry-standard color profile specification defined by the International Color Consortium.

The components of a color management system include

- collections of color characteristics (these collections are given various names, such as color tags, precision transforms, or in the case of the International Color Consortium, profiles)
- a color management module (CMM) that performs the color matching among color characteristic collections
- a programming interface for invoking color matching
- device-independent color spaces

A color management system gives the user the ability to perform color matching, to see in advance what colors cannot be accurately reproduced on a specific device, to simulate the range of colors of one device on another, and to calibrate peripheral devices using a device profile and a calibration application.

The next section describes the ColorSync Manager, which constitutes the color management system for the Macintosh OS.

About the ColorSync Manager

This section describes the ColorSync Manager architecture and how your application or device driver can use it for color conversion, color matching, color gamut checking, profile management, and creating CMMs that perform these services.

The ColorSync Manager allows your application or driver to maintain consistent color across devices and across platforms. The ColorSync Manager also let your users perform quick and inexpensive color proofing and see in advance what colors cannot be printed on their printers.

The ColorSync Manager consists of a collection of functions that your application or device driver can use to provide ColorSync support. These functions also allow you to create and manage profiles and to create CMMs that respond to requests from ColorSync-supportive applications or device drivers. The ColorSync Manager also includes a collection of display device profiles for Apple monitors and a robust default CMM.

Instead of providing your own device profile, device driver developers and peripheral manufacturers can obtain profiles from a number of vendors who provide them. For a list of profile vendors, contact the Developer Support organization of Apple Computer, Inc. See the Preface of this book for information explaining how to contact Developer Support.

ColorSync Manager shared library version number for PowerPC-based computers

The ColorSync Manager is implemented as a shared library on PowerPC[™]-based computers. The ColorSync Manager shared library version number is 0x02000000. ♦

Programming Interfaces

The ColorSync Manager programming interface allows your application to handle such tasks as color matching, color conversion, profile management, profile searching and accessing, reading individual tagged elements within a profile, embedding profiles in documents, modifying profiles, and creating CMMs that respond to requests for color matching and profile data transfer from one format to another.

About the ColorSync Manager's Memory Allocation and Use

The ColorSync Manager attempts to allocate the memory it requires from the following sources in this order:

- The current heap zone. If the current heap zone is set to the application heap, the ColorSync Manager will attempt to allocate the memory it requires from the application heap.
- The system heap. If the current heap zone is set to the system heap, the ColorSync Manager will try the system heap first and never attempt to allocate memory from the application heap.
- The Process Manager temporary heap. (If this final source does not satisfy the ColorSync Manager's memory requirements, any attempt to load the ColorSync Manager will fail.)

An application commonly sets the current heap zone to the application heap. When the ColorSync Manager is used apart from QuickDraw GX, this scenario commonly prevails, making application heap memory available to the ColorSync Manager.

However, QuickDraw GX holds a covenant with applications committing not to allocate memory from the application heap. QuickDraw GX sets the current heap zone to the system heap. Consequently, when the ColorSync Manager is used with QuickDraw GX, the ColorSync Manager is prohibited from allocating memory it requires from the application heap and must allocate all the memory it requires from the system heap and the Process Manager temporary heap.

Profiles

To perform color matching or color conversion across different base family color spaces requires the use of a profile for each device involved. Profiles describe various color characteristics. Profiles provide the ColorSync Manager with information necessary to convert color between device-dependent color spaces and device-independent color spaces. A profile may contain such information as lightest and darkest possible tones (referred to as white point and black point) and maximum densities for red, green, blue, cyan, magenta, and yellow. Together these measurements represent a color gamut.

For the ColorSync Manager, a profile consists of color data that follows the International Color Consortium (ICC) profile format. The International Color Consortium defines several different types of profiles. Each of these types of

Introduction to the ColorSync Manager

profiles must include a different required set of information, but all of these profile types follow the same format.

This format provides a single cross-platform standard for translating color data across devices and across operating systems. A profile created for a particular device is usable on systems running different operating systems. The founding members of the ICC include Adobe Systems Inc.; Agfa-Gevaert N.V.; Apple Computer, Inc.; Eastman Kodak Company; FOGRA (Honorary); Microsoft Corporation; Silicon Graphics, Inc.; Sun Microsystems, Inc.; and Taligent, Inc. These companies have committed to full support of this specification in their operating systems, platforms, and applications.

For information on how to obtain a copy of the *International Color Consortium Profile Format Specification*, version 2.0 document revision 3.x, contact the Developer Support organization of Apple Computer, Inc. See the preface of this book for information explaining how to contact Developer Support.

A **device profile** characterizes a particular device: that is, it describes the characteristics of a color space for a physical device in a particular state. A display, for example, might have a single profile, whereas a printer might have a different profile for each paper type or ink type it uses. Use of different paper types and ink types constitutes different printer states. When your application uses the ColorSync Manager to match colors between devices such as a display and a printer, it specifies the profile for each device when calling a ColorSync Manager color-matching function.

Device profiles are divided into three broad classifications:

- input devices such as scanners and digital cameras
- display devices such as monitors and flat-panel screens
- output devices such as printers and film recorders

A **color space profile** contains the data necessary to translate color values, such as CIE into RGB or RGB into CIE, as necessary for color matching. The ColorSync Manager, for example, uses color space profiles when mapping colors between different color spaces. Color space profiles provide a convenient means for CMMs to convert between different nondevice profiles.

Abstract profiles allow applications to perform special color effects independent of the devices on which the effects are rendered. For example, your application may choose to implement an abstract profile that increases yellow hue on all devices. Abstract profiles allow users of your application to

make subjective color changes to images or graphics objects by transforming the color data within the profile connection space.

A **device-linked profile** combines multiple profiles, such as various device profiles associated with the creation and editing of an image. A device-linked profile can include profile types other than device profiles, such as abstract profiles and color space profiles. The first and last profiles in the set are commonly those of the source and destination devices.

Profiles can reside in stand-alone files in the ColorSync™ Profiles folder, which exists in the Preferences folder inside the System Folder. Hardware vendors should have their users install profiles for their peripheral devices in the ColorSync™ Profiles folder.

Profiles can also be embedded within images. For example, profiles can be embedded in PICT, EPS, and TIFF files and in the private file formats used by applications. Embedded profiles allow for the automatic interpretation of color information as the color image is transferred from one device to another.

Note

The ICC profile format implemented in the ColorSync Manager is significantly different from the ColorSync 1.0 profile implementation. As implemented in the ColorSync Manager, a version 2.0 profile is a tagged-element structure. The extension infrastructure design supports this change. The profile supports use of lookup table transforms. The ColorSync 1.0 profile is memory resident. ♦

Profile Properties

Profiles can have different kinds of information in them. Recall that different types of profiles, such as a profile for a scanner or a profile for a printer, have different sets of minimum required tags and their element data. However, all profiles have at least a header followed by a required element tag table. The required tags may represent lookup tables, for example. The required tags for various profile types are described in the *International Color Consortium Profile Format Specification*.

Profiles contain additional information, such as a specification for how to apply matching (see the next section, “Color Management Modules”). Profiles may also have a series of optional and private tagged elements. These private tagged elements may contain custom information used by particular color

management modules. It is important to note that private tags limit the cross-platform portability of a profile and that, for this reason, Apple Computer, Inc. discourages use of them. The ColorSync Manager uses profiles that follow the format defined by the International Color Consortium. See the *International Color Consortium Profile Format Specification* for more information.

Color Management Modules

A color management module (CMM) uses profiles to convert and match a color in a given color space on a given device to or from another color space or device, perhaps a device-independent color space.

The ColorSync Manager includes color conversion functions that allow your application or driver to convert colors between color spaces belonging to the same base families without the use of CMMs; CMMs themselves can also call these color conversion functions. Color conversion and color matching across color spaces belonging to different base families always entail the use of a CMM.

When colors consistent with one device's gamut are displayed on a device with a different gamut, as in Figure 3-6 on page 3-16, a CMM attempts to minimize the perceived differences in the displayed colors between the two devices. The CMM does this by mapping the out-of-gamut colors into the range of colors that can be produced by the destination device.

The CMM uses lookup tables and algorithms for color matching, previewing color reproduction capabilities of one device on another, and checking for colors that cannot be reproduced. Although the ColorSync Manager includes an Apple-supplied CMM, used as the default CMM, peripheral developers can create their own CMMs, tailored to the specific requirements of their device.

Each profile header includes a field that names the preferred CMM to be used for performing color matching involving that profile. If two profiles used in a color-matching session name different CMMs, the ColorSync Manager follows an algorithm, described in "When Color Matching Occurs" on page 3-24, to determine the CMM to use.

The structure used to create a device-linked profile, which can contain many profiles, includes a field that identifies the CMM to be used for the entire color-matching session across all profiles.

Rendering Intents

Rendering intent refers to the approach taken when a CMM maps or translates the colors of an image to the color gamut of a destination device. The ICC version 2.0 profile specification defines a tag for each of the four supported rendering intents. The following four rendering intents supported by the ColorSync Manager allow the user to select an approach that best maintains the important aspects of the image:

- **Perceptual matching.** In this approach, all the colors of a given gamut may be scaled to fit within another gamut. The colors maintain their relative positions, so the relationship between colors is maintained. With realistic images such as scanned photographs, perceptual matching produces better results than colorimetric matching in most cases; in Figure 3-6, for example, the eye could compensate for the difference between the two gamuts, and a perceptually matched image on the printer device would look very similar to the original image on the RGB device. A disadvantage of perceptual matching is that all of the original colors are changed in the copy.
- **Relative colorimetric matching.** In this approach, colors that fall within the gamuts of both devices are left unchanged. For example, to match an image from the RGB gamut onto the printer gamut in Figure 3-6, only the colors in the RGB gamut that fall outside the printer gamut are altered. Relative colorimetric matching allows some colors in both images to be exactly the same, which is useful when colors must match quantitatively. A disadvantage of relative colorimetric matching is that many colors may map to a single color, resulting in tone compression. All colors outside the printer gamut in Figure 3-6, for example, would be converted to colors at the edge of its gamut, reducing the total number of colors in the image and possibly greatly altering its appearance. In relative colorimetric matching, colors outside the gamut are usually converted to colors with the same lightness, but different saturation, at the edge of the gamut. The final image may be lighter or darker overall than the original image, but the blank areas will coincide.
- **Saturation matching.** In some computer graphics, such as bar graphs and pie charts, the actual color displayed is less important than its vividness. In this approach, the relative saturation of colors is maintained from gamut to gamut. Colors outside the gamut are usually converted to colors with the same saturation, but different lightness, at the edge of the gamut.
- **Absolute colorimetric matching.** This approach is based on a device-independent color space in which the result is an idealized print viewed on a perfect paper having a large dynamic range and color gamut. In

reality paper cannot reproduce densities less than a particular minimum density (D_{min}). Absolute colorimetric matching leads to a close appearance match over most of the tonal range, but if the minimum density of the idealized image is different from that of the output image, the areas of the image that are left blank will be different. Colors that fall within the gamuts of both devices are left unchanged. Colors with densities that fall outside the dynamic density range of the destination device are clipped. While an appearance match may be achieved, there will be a loss of detail in some regions.

When Color Matching Occurs

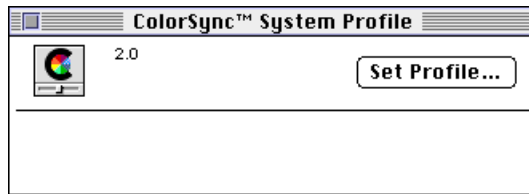
When a ColorSync-supportive scanning application creates a scanned image, it embeds a profile for the scanner driver in the image. The profile that is associated with the image and describes the characteristics of the device on which the image was created is called the **source profile**. If the colors in the image are subsequently converted to another color space by the scanning application or by another ColorSync-supportive application, the ColorSync Manager uses that source profile to identify the original colors and to match them to colors expressed in the new color space.

To display the image requires using another profile, which is associated with the output device, such as a display. The profile for that device is called the **destination profile**. If the image is destined for a display, the ColorSync Manager uses the display's profile (the destination profile) along with the image's source profile to match the image's colors to the display's gamut. If the image is printed, the ColorSync Manager uses the printer's profile to match the image's colors to the printer, including generating black and removing undercolors where appropriate.

If the color gamut of the source profile is different from the color gamut of the destination profile, the ColorSync Manager relies on the CMM and the information stored in both profiles for mapping the colors from the source profile's gamut to the destination profile's gamut. The CMM contains the necessary algorithms and lookup tables to enable consistent color among devices.

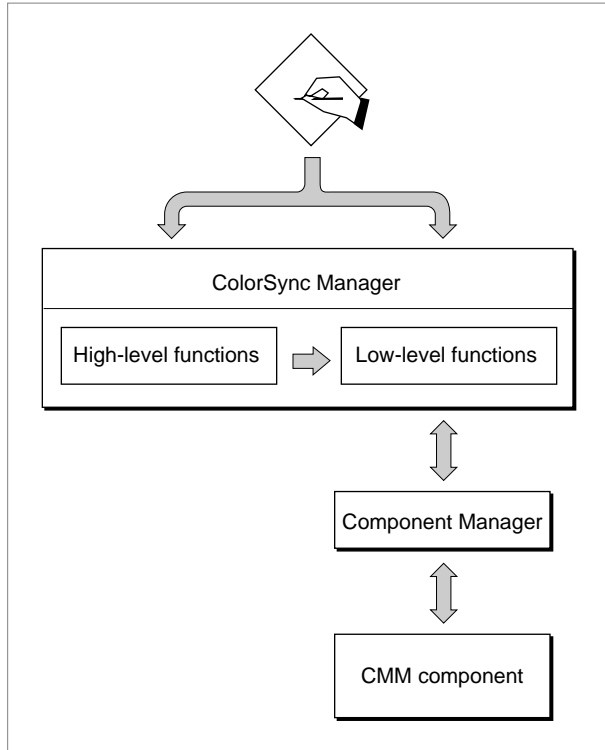
When using the ColorSync Manager functions for color matching, you specify the source and destination profiles. If you do not specify the source profile or the destination profile, the ColorSync Manager substitutes the system profile. Recall that the system profile is the profile for the system's display device and that the user can configure the profile to be used. Figure 3-7 shows the ColorSync™ System Profile control panel, which allows the user to configure the system profile to match the display device.

Figure 3-7 The ColorSync™ System Profile control panel



If the user does not specify a system profile through the ColorSync Manager control panel, the default system profile is used. The **default system profile** is the device profile for the Apple 13-inch color display.

Color matching between the source and destination color spaces happens inside the color management module component. Figure 3-8 shows the relationship between your application, the ColorSync Manager, the Component Manager, and the CMM component.

Figure 3-8 The ColorSync Manager and the Component Manager

QuickDraw GX and the ColorSync Manager

Unless your application uses QuickDraw GX to create and render images, your application must call ColorSync functions, such as `NCMBeginMatching` and `NCMDrawMatchedPicture`, to match colors between devices.

However, if your application uses QuickDraw GX and your application sets the view port attribute `gxEnableMatchPort`, the ColorSync Manager automatically matches colors when your application draws to the screen.

QuickDraw GX color profile objects contain ColorSync profiles, and each profile specifies the kind of matching that should be performed with it. For more information about QuickDraw GX color architecture, see the chapter “Colors and Color-Related Objects” in *Inside Macintosh: QuickDraw GX*.

QuickDraw GX version 1.1.2 or earlier uses ColorSync 1.0. However, because the ColorSync Manager provides robust backward compatibility including continued support of the ColorSync 1.0 API, you can use the ColorSync Manager with QuickDraw GX. For more information about the ColorSync Manager's backward compatibility, see the appendix, "ColorSync Manager Backward Compatibility."

What Users Can Do With ColorSync-Supportive Applications

Your ColorSync-supportive application or device driver can provide users with many features to help them reproduce color consistently across devices and across time and therefore reduce iterative color-proofing and color output surprises.

This section provides an overview of some of the features you can provide. How to implement these features is described in the chapters "Developing ColorSync-Supportive Applications" and "Developing ColorSync-Supportive Device Drivers."

Display Matching

When the user of a ColorSync-supportive application opens a file in which one or more ColorSync profiles have been embedded, the user benefits from display matching: that is, the user experiences consistent color from one display to another. If a color cannot be reproduced on the destination device and your program uses the ColorSync Manager API, the ColorSync Manager can map the color to the color gamut of the device.

Your application or driver should allow the user to embed or tag color-matching information and be able to display a tagged picture using the ColorSync Manager. Most importantly, your application must preserve picture comments in documents and allow the information to be passed on to the destination device.

Gamut Checking

Because not all colors can be rendered on all devices, you may want your application to warn users when a color they choose is out of gamut for the currently selected destination device. For example, you can use gamut checking to see if a given color is reproducible on a particular printer. If the color is not directly reproducible—that is, if it is out of gamut—you could alert

Introduction to the ColorSync Manager

the user to that fact. The ColorSync Manager provides the `CWCheckPixelFormat` and `CWCheckColors` functions for checking a color against a device's profile to see if it is in or out of gamut for the device. Your application should then display the results of this check in a window for the user.

Soft Proofing

Using the destination device's profile, your application can enable users to preview what a color image would look like on that device. This simulation of a device's output can save the user considerable time and cost.

Device-Linked Profiles

Most users use the same device configuration for scanning, viewing, and printing over a period of time. Your application can allow users to create a device-linked profile. A device-linked profile is a means of storing in a single profile a series of linked profiles that correspond to a specific configuration in the order in which the devices in that configuration are normally used. A device-linked profile represents a one-way link or connection between devices. A device-linked profile cannot be embedded into images.

Calibration

Your application can provide calibration services. A calibration application offers the option of calibrating a peripheral device based on a standard state or calibrating the device based on its current state.

If a peripheral device, such as a color printer, has drifted from its original state over time, a calibration application can make use of the characterization data contained in the corresponding profile to bring the color response back into range.

A user may want to improve the reproduction quality of a device without returning the device to a standard state. A profile can be created based on the current state of the device. The new profile is then used to calibrate that device. This approach to calibration maintains the existing dynamic density range while improving the device's overall quality.

Developing ColorSync-Supportive Applications

Contents

About ColorSync Application Development	4-4
About the ColorSync Manager Programming Interface	4-4
What Should a ColorSync-Supportive Application Do?	4-5
At a Minimum	4-5
Storing and Handling Profiles	4-6
How the ColorSync Manager Selects the CMM to Be Used	4-7
Developing Your ColorSync-Supportive Application	4-12
Determining If the ColorSync Manager Is Available	4-14
Providing Minimal ColorSync Support	4-15
Obtaining Profile References	4-16
Opening a Profile and Obtaining a Reference to It	4-17
Identifying the Current System Profile	4-19
Matching Colors to Displays Using ColorSync With QuickDraw Operations	4-20
Matching Colors in a Picture Containing an Embedded Profile	4-21
Matching Colors as Your User Draws a Picture	4-22
Setting a Large Profile Element	4-24
Creating a Color World for Color Matching and Checking Using the Low-Level Functions	4-27
Matching Colors Using the Low-Level Functions	4-29
Matching the Colors of a Pixel Map to the Display's Color Gamut	4-30
Matching the Colors of a Bitmap Image to the Display's Color Gamut	4-31
Embedding Profiles in Documents and Pictures	4-34
Extracting Profiles Embedded in Pictures	4-38
Step 1: Counting the Profiles in the PICT File	4-40
Step 2: Extracting the Profile	4-42

Searching for Profiles in the ColorSync™ Profiles Folder	4-49
Checking Colors Against a Destination Device's Gamut	4-51
Creating and Using Device-Linked Profiles	4-53
Considerations	4-56
Providing Soft Proofs	4-56
Calibrating a Device	4-58
Summary of the ColorSync Manager	4-59
Constants	4-59
Data Structures	4-63
Functions	4-69

This chapter describes how you can use the ColorSync Manager to provide your users with color-matching and color gamut-checking services. Your ColorSync-supportive application can match the colors of an image created using one device to the color gamut of another device on which the image is to be rendered. Your application can allow your users to preview the results and adjust the colors of the matched image if desirable. To match or check the colors of an image across devices with different gamuts and characteristics, the ColorSync Manager uses profiles that contain information depicting a device. Using the ColorSync Manager, your application can search for and display a list or menu containing the names of available profiles for a particular device type, such as a printer, and allow your users to select the profile to be used to define the destination device. You can modify the content of profiles to change how an image is to be rendered. These and other features that your application can provide using the ColorSync Manager are described in this chapter.

You need to read this chapter if your application will support the ColorSync Manager and provide your users with various color-matching and color-checking features. You should read this chapter even if you will provide only minimal support allowing your users to modify documents containing color images created using other applications that fully support the ColorSync Manager. This chapter tells you how to preserve profiles embedded in documents along with the images with which they are associated.

The features described in this chapter can also be used in developing ColorSync-supportive device drivers. Therefore, you should read this chapter in addition to the chapter “Developing ColorSync-Supportive Device Drivers” if you are developing a device driver that supports the ColorSync Manager.

Before you read this chapter, you should read the chapter “Introduction to the ColorSync Manager” in this book. The introductory chapter explains color theory and color management systems (CMSs). It provides an overview of the ColorSync Manager CMS, including the use of profiles. “Introduction to the ColorSync Manager” also explains key terms that describe aspects of the ColorSync Manager, which are used throughout this chapter but not defined again in it.

While reading this chapter, you might want to refer to the chapter “ColorSync Manager Reference for Applications and Device Drivers” in the *Advanced Color Imaging Reference* for details related to functions this chapter discusses.

Most high-level applications that support the ColorSync Manager automatically either use the color management module (CMM) specified by the source profile or use the Apple-supplied CMM, as determined internally by the

CMM selection algorithm described later in this chapter. For this reason, you do not need to read the two ColorSync Manager chapters addressed to developers who provide CMMs—“ColorSync Manager Reference for Color Management Modules” in the *Advanced Color Imaging Reference* on the enclosed CD and “Developing Color Management Modules” in this book—nor do you need to read the chapter “Developing ColorSync-Supportive Device Drivers” in this book. However, you should read the appendix, which describes ColorSync Manager 2.0 backward compatibility.

About ColorSync Application Development

The ColorSync Manager provides your application with color-matching capabilities that your users can employ without the need for a proprietary environment. The ColorSync Manager provides the first system-level implementation of an industry-standard color-matching system. Because the ColorSync Manager supports the version 2.0 profile format defined by the International Color Consortium (ICC), a color image your user creates can be color matched, rendered, and modified by another user running another application on another platform that supports the version 2.0 profile format. Conversely, your application can modify and color match images created by other applications that support the ColorSync Manager or a CMS that includes support for the version 2.0 profile.

The ColorSync Manager requires Color QuickDraw and System 7.0 or later. The Component Manager is packaged with the ColorSync Manager and installed at startup in systems that do not include it (that is, systems prior to 7.1).

About the ColorSync Manager Programming Interface

The ColorSync Manager programming interface allows your application to handle such tasks as color matching, color conversion, profile management, profile searching and accessing, reading individual tagged elements within a profile, embedding profiles in documents, and modifying profiles. The ColorSync Manager includes the five interface files for developers that you can use for either 68020 or later or PowerPC development. Of the five interface files, your application must include at least the following three:

`CMApplication.h`

Interface to the ColorSync Manager functions and data types for applications and device drivers.

`CMConversions.h`

Interface for conversions to and from base-derived color spaces.

`CMICCPProfile.h`

Definitions for the version 2.0 profile for profile developers.

This interface file contains enumerations and structures, such as the version 2.0 profile header, that your application requires.

Therefore, you must include it.

What Should a ColorSync-Supportive Application Do?

Your ColorSync-supportive application can provide a rich set of color-matching features. Your application can color match images, pixel maps, bitmaps, and even individual colors. In addition to color matching, you can handle such tasks as color conversion, color gamut checking, soft proofing of images, profile management, profile searching and accessing, reading individual tagged elements within a profile, embedding profiles in documents, extracting embedded profiles, and modifying profiles.

Your application can provide an interface that offers your user selection menus allowing the user to choose how an image is to be rendered and the profile to associate with an image. It can show the user the colors of an image that are in or out of gamut for a particular device on which the image is to be produced and how the ColorSync Manager adjusts for colors that are out of gamut. This allows the user to preview differences that occur in the color-matching transition between gamuts and make corrections if necessary.

At a Minimum

The ColorSync Manager allows your application to preserve high fidelity to the original colors of an image—whether the image was created using your application or another—by supporting the use of embedded profiles. Your application can take advantage of a profile embedded along with an image, matching the original colors of the device used to create the image to those of the destination display or printer. Even if your application doesn't support some of the more advanced features the ColorSync Manager affords, such as soft proofing, you should color match images using the source profile, if one is identified and available.

At a minimum, your application should preserve images tagged with a profile by not stripping out picture comments used to embed profiles or by leaving profiles in documents that use other methods to include them.

It is important for your application to tag an image with the profile for the device used to create the image and to preserve existing tagging because a picture that is not tagged assumes use of the system profile. If the picture is moved to a different system that uses a different system profile, the picture will be drawn differently. “Providing Minimal ColorSync Support,” beginning on page 4-15, explains how to preserve embedded profiles, and “Embedding Profiles in Documents and Pictures,” beginning on page 4-34 explains how to tag an image. Some of these features are described in greater detail in the rest of this chapter.

Storing and Handling Profiles

Profiles for use with the ColorSync Manager are stored in the ColorSync™ Profiles folder within the Preferences folder of the System Folder. When you install the ColorSync Manager, the ColorSync™ Profiles folder contains a selection of display profiles for all Apple color monitors.

The ColorSync Manager provides a control panel, the ColorSync™ System Profile control panel, to allow the user to select the profile corresponding to the system’s display. This profile then becomes the system profile. Your application specifies the profiles to be used for color matching when the application calls a ColorSync Manager function. For most functions, the ColorSync Manager uses the system profile as the default profile if your application doesn’t specify a profile. Some functions require that you explicitly specify a profile by reference.

Device drivers for ColorSync-supportive input and output devices, such as scanners and printers, may install the profiles they use in the ColorSync™ Profiles folder, making them available to your application for color matching or gamut checking. If your application creates device-linked profiles, described on page 4-53, you should place them in the ColorSync™ Profiles folder.

Your application can provide the interface to allow your user to choose a profile for a specific device. Using the ColorSync Manager functions, your application can search the ColorSync™ Profiles folder and display information about available profiles to your user.

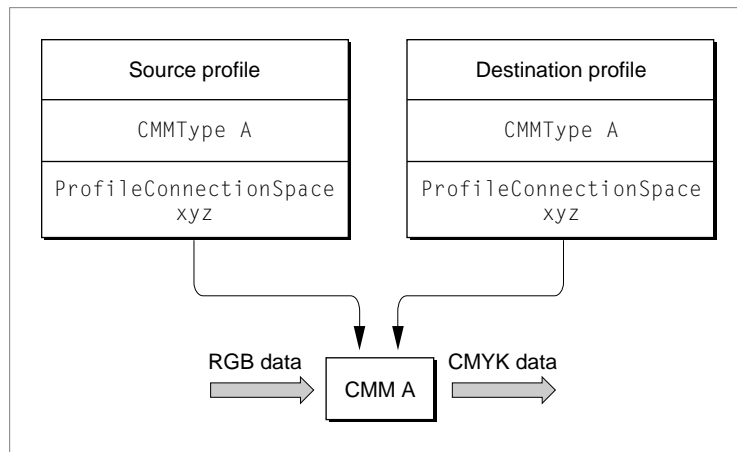
These features are described in greater detail in the rest of this chapter. As described in “Providing Minimal ColorSync Support” on page 4-15, your application should, at a minimum, leave profile information intact in the documents and pictures that it imports or copies into its own documents.

How the ColorSync Manager Selects the CMM to Be Used

A profile header contains a field called the `CMMType` field that specifies the preferred CMM for that profile. When the source and destination profiles specify different CMMs or when a specified CMM is unavailable or unable to provide a requested color-matching service, the ColorSync Manager follows a CMM selection algorithm or arbitration scheme to determine which CMM to use for color conversion and matching. Here is how the CMM selection algorithm works:

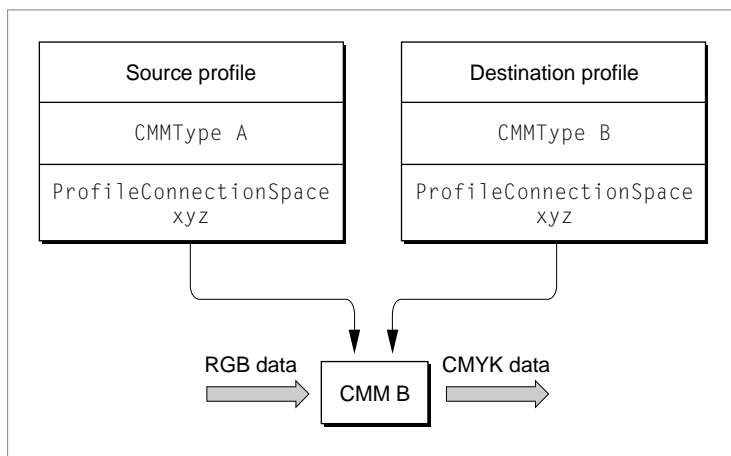
- If the source and destination profiles specify the same CMM and that CMM component is available and able to perform the matching, then the specified CMM maps the colors directly from the color space of the source profile to the color space of the destination profile. This is the simplest scenario, and Figure 4-1 illustrates it.

Figure 4-1 Color matching when the source and destination profiles specify the same CMM



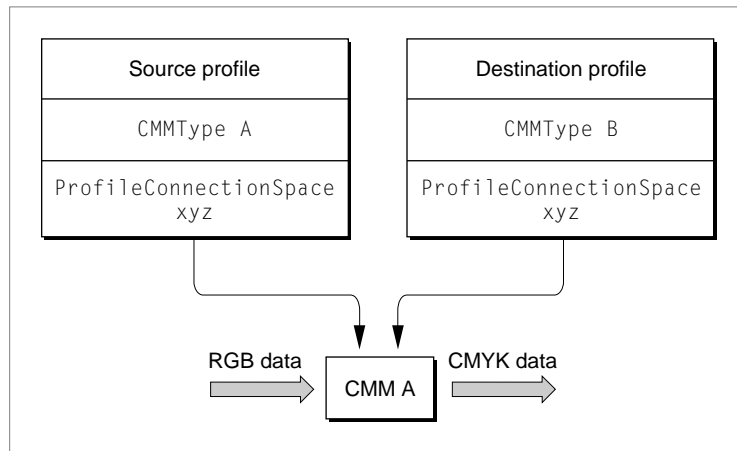
- If the source and destination profiles specify different CMMs, then the ColorSync Manager follows these steps to identify the CMM to use:
 1. If the CMM specified by the destination profile is available, is able to perform the color matching using the two profiles, and is not the Apple-supplied default CMM, then the ColorSync Manager uses this CMM. Figure 4-2 shows this scenario.

Figure 4-2 Color matching using the destination profile's CMM



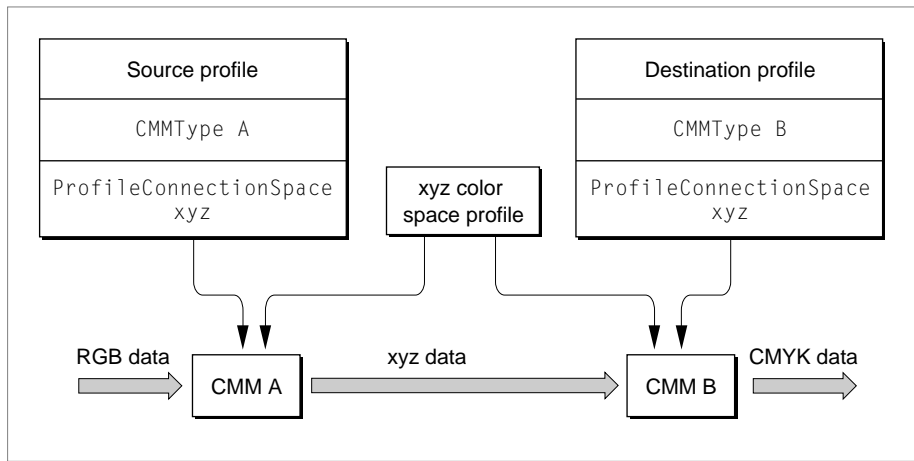
2. If the destination profile's preferred CMM is unavailable or unable to perform the color-matching request using the two profiles, then the ColorSync Manager looks for the CMM specified by the source profile. If the CMM specified by the source profile is available, is able to perform the color matching using the two profiles, and is not the Apple-supplied default CMM, the ColorSync Manager uses this CMM. Figure 4-3 shows this scenario.

Figure 4-3 Color matching using the source profile's CMM



3. If both the source-preferred CMM and the destination-preferred CMM are available, but neither is able to perform the match alone, the ColorSync Manager uses the source profile's CMM to convert the colors of the source image from the source profile's color space to an interchange color space using the XYZ color space profile as the destination profile. Next, the ColorSync Manager uses the preferred CMM specified by the destination profile to convert the colors now specified in the interchange color space to colors expressed in the color space of the destination profile using the XYZ color space profile as the source profile. The color conversion and matching work this way if both profiles specify the same interchange color space. Figure 4-4 shows this scenario.

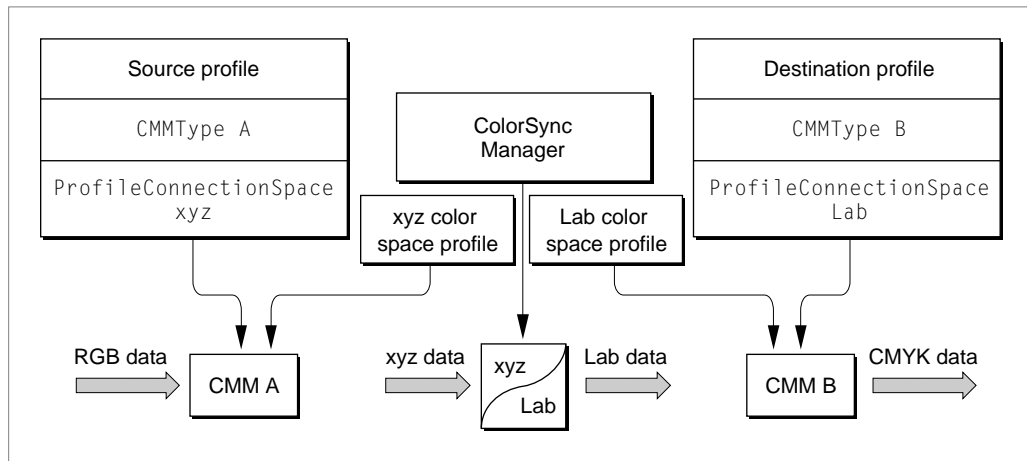
Figure 4-4 Color matching through an XYZ interchange space using both CMMs



4. If both the source-preferred CMM and the destination-preferred CMM are available, but neither is able to perform the match alone and both profiles specify different interchange color spaces, the ColorSync Manager uses the source profile's CMM to convert the colors of the source image from the source profile's color space to its interchange color space using the appropriate color space profile as the destination profile. The example shown in Figure 4-5 uses the XYZ color space profile as the destination profile. Then the ColorSync Manager inserts a part into the process, itself converting colors from the source profile's interchange color space to the

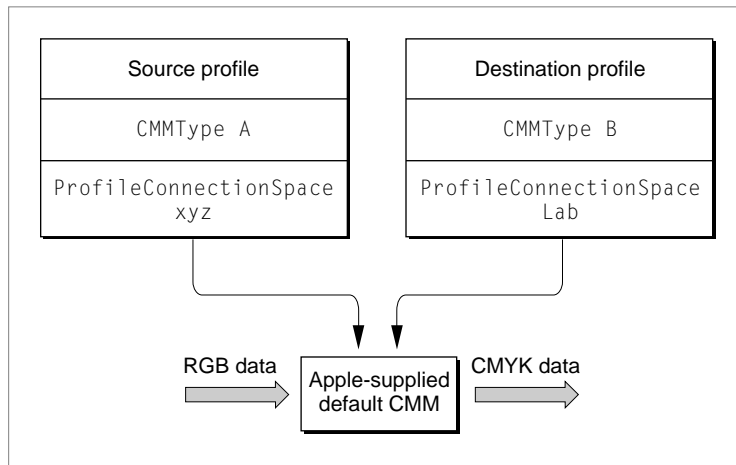
destination profile's interchange color space. Next, the ColorSync Manager uses the preferred CMM specified by the destination profile to convert the colors now specified in the destination profile's interchange color space to colors expressed in the destination profile's color space using the appropriate color space profile as the source profile. The example shown in Figure 4-5 uses the Lab color space profile as the source profile.

Figure 4-5 Matching using both CMMs and two interchange color spaces



- If neither the source nor the destination profile's preferred CMM is available or able to perform the color conversion and matching, then the ColorSync Manager uses the Apple-supplied default CMM, which will always attempt to perform the match. Figure 4-6 shows this scenario.

Figure 4-6 Color matching using the Apple-supplied default CMM



Developing Your ColorSync-Supportive Application

This section describes some of the tasks your application can perform to implement the color-matching and color-checking features you can provide using the ColorSync Manager functions.

This section explains how to

- determine if version 2.0 of the ColorSync Manager is available on a 68K-based system or a PowerPC-based system, described on page 4-14
- provide minimal ColorSync support by preserving embedded profiles in existing documents, described on page 4-15

- obtain a unique reference to a profile, which provides access to the profile, described on page 4-16
- obtain the name of the current system profile and other information about the profile, described on page 4-19
- match the colors of an image to the system's display using ColorSync with QuickDraw operations, described on page 4-20
- create a color world comprised of profiles required for a color-matching or gamut-checking session using the low-level ColorSync Manager functions that do not rely on QuickDraw, described on page 4-27
- match the colors of an image to the system's display without using QuickDraw, described on page 4-29
- store a profile in a document containing the image created using the profile, referred to as embedding a profile, described on page 4-34
- extract an embedded profile from a document in order to use it, described on page 4-38
- search through the ColorSync™ Profiles folder for profiles that meet certain criteria, described on page 4-49
- check the colors of an image against the color gamut of the device for which it is destined, such as a printer, described on page 4-51
- create and use a special type of profile called a device-linked profile containing a number of concatenated profiles used in sequence, described on page 4-53
- provide soft proofs, described on page 4-56
- calibrate a device, described on page 4-58

Determining If the ColorSync Manager Is Available

To determine whether version 2.0 of the ColorSync Manager is available on a 68K-based Macintosh system, you use the `Gestalt` function with the `gestaltColorMatchingVersion` selector. You can modify and use the following sample code to test for the presence of the ColorSync Manager. This function initializes the Boolean `ColorSyncAvailable` variable to `false` and sets it to `true` if version 2.0 or later of the ColorSync Manager is installed.

```
{
    Boolean ColorSyncAvailable = false;
    long    version;
    if (Gestalt(gestaltColorMatchingVersion, &version) == noErr)
        { if ( version >= gestaltColorSync20) }
          { (ColorSyncAvailable = true; }
}
```

For a PowerPC-based system, to determine if the ColorSync Manager shared libraries have been loaded, use the `Gestalt` function with the `gestaltColorMatchingAttr` selector. Test the bit field (bit 1) indicated by the `gestaltColorMatchingLibLoaded` constant in the response parameter. If the bit is set, the ColorSync Manager shared libraries are loaded. You can use the following sample code to determine if the ColorSync Manager is available on a PowerPC-based system. This code fragment initializes the `colorSyncAvailable` Boolean variable to `false`.

```
Boolean CheckIfColorSyncAvailableOnPPC(void)
{
    Boolean ColorSyncAvailable = false;
    long    gestaltResponse;
    if (TrapAvailable(_Gestalt))
        { ColorSyncAvailable =
          ((Gestalt(gestaltColorMatchingAttr, &gestaltResponse) = noErr)
           && (gestaltResponse & (1 << gestaltColorMatchingLibLoaded)));
        }
}
```

Providing Minimal ColorSync Support

The ICC's open architecture describes the profile format that all compliant CMSs must support. The common profile format allows one user to electronically transfer a document containing a color image to another user with the assurance that the original image will be rendered faithfully according to the source profile specification through use of a CMS such as the ColorSync Manager.

To ensure this, the application or driver used to create the image stores the profile for the source device in the document containing the color image. The application can do this automatically or allow the user to tag the image. If the source profile is embedded within the document, a user can move the document from one system to another without concern for whether the intended recipient has the profile used to create the image installed on his or her computer.

To support the ColorSync Manager, your application should, at a minimum, leave profile information intact in the documents and pictures it imports or copies. That is, your application should not strip out profile information from documents or pictures created with other applications. Even if your application does not use profile information, your users may be able to take advantage of it when using the documents or pictures with other applications.

For example, profiles may be embedded in pictures that your users paste into your application. Profiles can be embedded in formats such as PICT or TIFF files. For files of type 'PICT', the ColorSync Manager defines the following picture comments for embedding profiles and for performing color matching:

```
/* PicComment IDs */
enum {
    cmBeginProfile          = 220,
                                /* begin ColorSync 1.0 profile */
    cmEndProfile            = 221,
                                /* end a ColorSync 2.0 or 1.0
                                profile */
    cmEnableMatching        = 222,
                                /* begin color matching for either
                                ColorSync 2.0 or 1.0 */
    cmDisableMatching       = 223,
                                /* end color matching for either
                                ColorSync 2.0 or 1.0 */
    cmComment               = 224
```

Developing ColorSync-Supportive Applications

```

/* embedded ColorSync 2.0 profile
   information */

};

/* PicComment selectors for cmComment */
enum {
    cmBeginProfileSel          = 0,    /* begining of a ColorSync 2.0
                                         profile; profile data to
                                         follow */
    cmContinueProfileSel       = 1,    /* continuation of a ColorSync
                                         2.0 profile; profile data to
                                         follow */
    cmEndProfileSel            = 2     /* end of ColorSync 2.0 profile
                                         data; no profile data
                                         follows */
};

```

The picture comment `kind` value of 224 is defined for embedded ColorSync Manager version 2.0 profiles. This picture comment is followed by a 4-byte selector that describes the type of data in the picture comment. Your application should leave these comments and the embedded profiles they define intact. Similarly, if your application imports or converts file types defined by other applications, your application should maintain the profile information embedded in those files, too.

Your application can also embed picture comments and profiles in documents and pictures it creates or modifies. For information describing how to do this, see “Embedding Profiles in Documents and Pictures” on page 4-34. *Inside Macintosh: Imaging With QuickDraw* describes picture comments in detail.

Obtaining Profile References

Most of the ColorSync Manager functions require that your application identify the profile or profiles to be used in carrying out the work of the function. For example, when your application calls functions to perform color matching or color gamut checking, you must identify the profiles to be used for the session. For high-level functions that use QuickDraw, you specify a source profile and a destination profile. For low-level functions, you specify a color world containing source and destination profiles or a set of concatenated profiles. You can also create a device-linked profile, which is described later in this chapter,

but to do so your application must first obtain references to all the profiles that will comprise the device-linked profile.

Your application must identify the profile to be used when you intend to modify the profile's contents—for example, you might modify a profile based on the rendering intent a user specifies—and to copy or verify a profile's elements.

The ColorSync Manager provides for multiple concurrent accesses to a single profile through use of a private data structure called a *profile reference*. A **profile reference** is a unique reference to a profile; it is the means by which your application identifies a profile and gains access to the contents of that profile. Many applications can use the same profile at the same time, each with its own reference to the profile.

Opening a Profile and Obtaining a Reference to It

To open a profile and obtain a reference to it, you call the `CMOpenProfile` function. (The `CMCopyProfile`, `CWNewLinkProfile`, and `CMNewProfile` functions also return profile references.) To identify a profile that is file based or memory based, you must give its location.

The ColorSync Manager defines the following data type that you use to tell the profile's location:

```
struct CMProfileLocation{
    short      locType;
    CMProfLoc  u;
};
```

The `CMProfileLocation` data type contains a member called `CMProfLoc` for which you specify a value using the following union defined by the ColorSync Manager:

```
union CMProfLoc {
    CMFileLocation      fileLoc;
    CMHandleLocation    handleLoc;
    CMPtrLocation       ptrLoc;
};
```

The `CMProfLoc` value provides the actual location of the profile. In most cases, a ColorSync profile is stored in a disk file, and you use the union to give the file

specification. However, to support special requirements, a profile can also be located in memory, or it may be a temporary profile, meaning that the profile will not persist in memory after your application uses it for a color session.

Your application uses a data type of `CMFileLocation` to provide a file specification for a profile stored in a disk file, a data type of `CMHandleLocation` to specify a handle for a profile stored in relocatable memory, and a data type of `CMPtrLocation` to specify a pointer to a profile stored in nonrelocatable memory.

To identify the kind of data type you assigned to the `u` field of `CMProfileLocation`, you assign to the `CMProfileLocation.locType` field one of the constants or numeric equivalents defined by the following enumeration:

```
enum {
    cmNoProfileBase = 0,
    cmFileBasedProfile= 1,
    cmHandleBasedProfile= 2,
    cmPtrBasedProfile= 3
};
```

For example, for a file-based profile, the `u` field would hold a file specification and the `locType` field would hold the constant `cmFileBasedProfile`. Your application passes a `CMProfileLocation` structure to the function when it calls the `CMOpenProfile` function and the function returns a reference to that profile.

Listing 4-1 shows an application-defined function, `MyOpenProfileFSSpec`, that assigns the file specification for the profile file to the `profLoc` union and identifies the location type as file based. Given the file specification, `MyOpenProfileFSSpec` then calls the `CMOpenProfile` function, passing to it the profile's file specification and receiving in return a reference to the profile.

Listing 4-1 Opening a reference to a file-based profile

```
CMError MyOpenProfileFSSpec (FSSpec spec, CMProfileRef *prof)
{
    CMError          cmerr;
    CMProfileLocation profLoc;

    profLoc.locType = cmFileBasedProfile;
    profLoc.u.file.spec = spec;
```

```

        cmerr = CMOpenProfile(prof, &profLoc);

        return cmerr;
    }

```

▲ WARNING

Problems using a single profile reference can occur when different processes within the same application use the same profile reference. For example, if your application allows multiple windows to remain open at the same time, you cannot determine which window a user might close first. If a user closes a window and the window's process disposes of the profile reference, the processes associated with the remaining open windows will no longer have access to the profile. The CSDemo sample application provided on the enclosed CD shows how to handle this kind of circumstance. ▲

Identifying the Current System Profile

For the `NCMBeginMatching`, `NCMDrawMatchedPicture`, and `NCWNewColorWorld` functions, your application can specify `NULL` to signify the system profile. For all other functions—for example, the `CMGetProfileElement` function, the `CMValidateProfile` function, and the `CMCopyProfile` function—for which you want to specify the system profile, you must give an explicit reference to the profile. You can use the `CMGetSystemProfile` function to obtain a reference to the system profile.

Each profile has a name associated with it, including the profile configured as the system profile. There are cases in which your application may need to display the name of the system profile to the user. For example, you may want to present a list or selection menu to your user showing the names of all available display profiles. To obtain the name of the system profile, your application can call the `CMGetScriptProfileDescription` function.

To call this function, your application must first obtain a reference to the system profile because you cannot specify `NULL` to identify it. Given a profile reference, the `CMGetScriptProfileDescription` function returns the profile name and script code. Listing 4-2 uses the `CMGetSystemProfile`

and `CMGetScriptProfileDescription` functions to obtain a reference to the system profile and the system profile's name and script code.

Listing 4-2 Obtaining the current system profile

```
void MyPrintSystemProfileName(void)
{
    CMError          cmErr;
    CMProfileRef      sysProf;
    Str255            profName;
    ScriptCode        profScript;

    cmErr = CMGetSystemProfile(&sysProf);

    if (cmErr == noErr)
    {
        cmErr = CMGetScriptProfileDescription(sysProf, profName,
                                              &profScript);

        if (cmErr == noErr)
        {
            DrawString(profname);
        }
    }
}
```

Matching Colors to Displays Using ColorSync With QuickDraw Operations

To provide your user with images and pictures showing consistent colors across displays, your application can use the ColorSync Manager to match the colors in the user's pictures and documents with the colors available on the user's current display. If a color cannot be reproduced on the current system's display, the ColorSync Manager maps the color to the color gamut of the display according to the specifications defined by the profiles.

The ColorSync Manager provides two high-level functions that use QuickDraw that your application can call to draw a color picture to the current display. One function, `NCMDrawMatchedPicture`, uses the source profile embedded in the picture to match the picture's colors to the display's gamut defined by the

system profile. The other function, `NCMBeginMatching`, uses the source and destination profiles you specify to match the colors of the source image to the colors of the device for which it is destined.

On all systems, the current display device's profile should be configured as the system profile. The ColorSync™ System Profile control panel allows the user to configure the current display's profile as the system profile. Because the ColorSync Manager recognizes the system profile as that of the current display, you can specify `NULL` to indicate the system profile instead of explicitly giving a profile reference. Passing `NULL` as a profile reference to the ColorSync Manager functions directs the ColorSync Manager to use the system profile.

The following subsections describe how to use these high-level matching functions, which automatically perform color matching in a manner acceptable to most applications. Listing 4-3 on page 4-23 shows sample code that performs color matching to a display in different ways using the high-level functions.

However, if your application needs a finer level of control over color matching, you can use the low-level ColorSync Manager color-matching functions, described in “Matching Colors Using the Low-Level Functions” on page 4-29 to match the colors of a bitmap, a pixel map, or a list of colors.

Matching Colors in a Picture Containing an Embedded Profile

If a user copies a picture that includes a profile into one of your application's documents, your application can use the ColorSync Manager's high-level function `NCMDrawMatchedPicture` to match the colors in that picture to the display on which you draw it.

The `NCMDrawMatchedPicture` function automatically matches all colors to the color gamut of the display device using the device's profile as the picture is drawn. To use this function, you identify only the profile for the display device. The function acknowledges color-matching picture comments embedded in the picture and uses embedded profiles. The source profile for the device on which the image was created should be embedded in the QuickDraw picture whose handle you pass to the function; the `NCMDrawMatchedPicture` function uses the embedded source profile, if it exists. If the source profile is not embedded, the function uses the current system profile as the source profile. A picture may have more than one profile embedded. If so, the `NCMDrawMatchedPicture` function will use the profiles successively if they are embedded correctly.

By specifying `NULL` as the destination profile when you use this function, you are assured that the system profile—that is, the profile for the main screen—is

used as the destination profile. Alternatively, your application can call the `CMGetSystemProfile` function to obtain a reference to the profile and specify the system profile explicitly; the portion of code that calls the `NCMDrawMatchedPicture` function in Listing 4-3 on page 4-23 handles use of the system profile this way.

Considerations

For embedded profiles to be used correctly, the currently effective profile must be terminated by a picture comment of kind `cmEndProfile` after drawing operations using that profile are performed. If a picture comment was not specified to end the profile, the profile will remain in effect until the next embedded profile is introduced with a picture comment of kind `cmBeginProfile`. However, use of the next profile might not be the intended action. It is good practice to always pair use of the `cmBeginProfile` and `cmEndProfile` picture comments. When the ColorSync Manager encounters an `cmEndProfile` picture comment, it restores use of the system profile for matching until it encounters another `cmBeginProfile` picture comment.

If your application allows a user to modify an image that you color-matched using the `NCMDrawMatchedPicture` function, your application must either embed the system profile in the picture file or convert and match the colors of the modified image to the colors of the source profile. The method you choose is specific to your application.

Matching Colors as Your User Draws a Picture

To use Color QuickDraw functions to draw a document with colors matched to a display, your application can simply use the `NCMBeginMatching` function before calling Color QuickDraw functions, and then conclude its drawing with the `CMEndMatching` function. Color QuickDraw drawing functions are described in *Inside Macintosh: Imaging With QuickDraw*.

To use this function, you must specify both the source and destination profiles. If the user creates the image using your application, you can specify `NULL` for both the source and destination profiles because the source and destination devices are the same, the current display. This function returns a reference to the color-matching session that you must pass to the `CMEndMatching` function to terminate color matching. The latter portion of code shown in Listing 4-3 demonstrates how to use the `NCMBeginMatching` and `CMEndMatching` pair of functions.

Listing 4-3 Two methods of color matching to a display

```

void MyMatchingToDisplays(void)
{
    CMError          cmErr;
    CMProfileRef      sysProf;
    CMProfileRef      targProf;
    PicHandle         hPICT;
    Rect              rcPICT;
    CMMatchRef        matchRef;

    /* use the system profile as the destination profile for the
       NCMDrawMatchedPicture function */

    cmErr = GetPict(&hPICT, &rcPICT);
    if (cmErr == noErr)
    {
        cmErr = CMGetSystemProfile(&sysProf);
    }

    if (cmErr == noErr)
    {
        NCMDrawMatchedPicture(hPICT, sysProf, &rcPICT);

        KillPicture(hPICT);
        (void) CMCloseProfile(sysProf);
    }

    /* use the system profile as the source profile and another profile as the
       destination profile for the NCMBeginMatching and CMEndMatching functions */

    cmErr = CMGetSystemProfile(&sysProf);

    if (cmErr == noErr)
    {
        cmErr = MyGetImageTargetProfile(&targProf);
    }

    if (cmErr == noErr)
    {
        cmErr = NCMBeginMatching(sysProf, targProf, &matchRef);
    }
}

```

```

    (void) CMCloseProfile(sysProf);
    (void) CMCloseProfile(targProf);
}

if (cmErr == noErr)
{
    MyDoDrawing();

    CMEndMatching(matchRef);
}
}

```

Setting a Large Profile Element

When you need to set a large amount of element data for a tag in a profile, you can use the `CMSetPartialProfileElement` function to copy the data to the profile in segments. If you know the size of the element data, it is good practice to call `CMSetProfileElementSize` to reserve space for the data before you set it. First setting the size avoids incurring the extensive overhead required to increase the size for the element data with each call to append another segment of data. In addition to reserving the element data size, the `CMSetProfileElementSize` function sets the element tag, if it does not already exist.

After you set the element size, you can call the `CMSetPartialProfileElement` function repeatedly, as many times as necessary, each time appending a segment of data to the end of the data already copied until all the element data is copied.

Listing 4-4 Setting the element size before setting the element data in segments

```

#include <Types.h>
#include <Memory.h>
#include <StandardFile.h>
#include <OSUtils.h>
#include <CMApplication.h>
#include "MacUtils.h"

/* test element signature */
#define kElemTag    'test'

```

Developing ColorSync-Supportive Applications

```

/* kElemSize must be evenly divisible by kNSegments */
#define kElemSize    10000
#define kNSegments   10

void main(void);

void main(void)
{
    CMError          err;
    StandardFileReply reply;
    CMProfileRef      prof;
    CMProfileLocation location;
    unsigned long     offset;
    unsigned long     byteCount;
    Ptr               p;
    long              i;

    /* for error handling */
    err = noErr;

    p   = NULL;

    /* standard Mac Init */
    InitMacintosh(8);

    /* request file spec for destination */
    StandardPutFile("\p", "\pnew profile", &reply);

    /* bail if user canceled */
    if (!reply.sfGood)
    {
        return;
    }
}

```

```

/* create new empty profile */
if (err == noErr)
{
    location.locType= cmFileBasedProfile;
    location.u.fileLoc.spec= reply.sfFile;

    err = CMNewProfile(&prof, &location);
}

/* allocate buffer for element data: cleared to 0s */
if (err == noErr)
{
    if ((p = NewPtrClear(kElemSize/kNSegments)) == NULL)
    {
        err = MemError();
    }
}

/* set total element size; also initially creates the element tag */
if (err == noErr)
{
    err = CMSetProfileElementSize(prof, kElemTag, kElemSize);
}

/* set element data in segments */
if (err == noErr)
{
    for (i = 0; i < kNSegments; i++)
    {
        offset = (i * (kElemSize / kNSegments));
        byteCount= (kElemSize / kNSegments);
        err = CMSetPartialProfileElement(prof, kElemTag, offset, byteCount, p);

        if (err != noErr)
            break;
    }
}

/* update profile disk file */
if (err == noErr)
{

```

```

    err = CMUpdateProfile(prof);
}

/* close profile */
if (err == noErr)
{
    err = CMCloseProfile(prof);
}

/* free buffer */
if (p!= NULL)
{
    DisposePtr(p);
}
}

```

Creating a Color World for Color Matching and Checking Using the Low-Level Functions

A color world is a reference to a private ColorSync structure that represents a unique color-matching session. Although profiles can be large, a color world is a compact representation of the mapping needed to match between profiles. Conceptually, you can think of a color world as a sort of “matrix multiplication” of two or more profiles that distills all the information contained in the profiles into a fast multidimensional lookup table.

For the ColorSync Manager low-level functions, a color world defines the aspects that characterize how the color-matching session will occur based on information contained in the profiles that you supply when your application sets up the color world. Your application can define a color world for color transformations between a source profile and a destination profile, or it can define a color world for color transformations among a series of concatenated profiles.

For the low-level ColorSync Manager functions, a color world is the equivalent of the ColorSync Manager QuickDraw-based functions’ source and destination profiles. From your application’s perspective, the difference in specifying profiles for the low-level functions is that instead of calling a function and passing it references to the profiles for the session, first you must create a color world using those profile references and pass the color world to the function.

Your application calls the `NCWNewColorWorld` function to set up a simple color world to be used for color transformations involving two profiles—a source profile and a destination profile—and the function returns a reference to the color world it creates. Setting up a color world to be used for color processing involving a series of concatenated profiles or a single device-linked profile, which contains a series of profiles, is slightly more complex. Here are the steps you take:

- 1. Obtain references to the profiles to be used for the concatenated color world.**

For information describing how to obtain references to the profiles for the color world, see “Obtaining Profile References” on page 4-16.

- 2. Set up an array containing references to the profiles comprising the set.**

Before your application calls the `CWConcatColorWorld` function to create the color world, you must establish the profile set. The ColorSync Manager defines the following data structure of type `CMConcatProfileSet` that you use to specify the profile set:

```
struct CMConcatProfileSet {
    unsigned short    keyIndex;
    unsigned short    count;
    CMProfileRef      profileSet[1];
};
```

Your application also uses the `CMConcatProfileSet` data structure to define a profile set for a device-linked profile. See “Creating and Using Device-Linked Profiles” on page 4-53 for more information.

Your application must create an array that contains references to the profiles for the color world, specifying these references in processing order. You must specify the number of profile references in the array as the value of the `CMConcatProfileSet.count` field, using a one-based number. You assign the profile array to the `CMConcatProfileSet.profileSet` field.

The ColorSync Manager defines rules governing the types of profiles you can specify in a profile array. These rules differ depending on whether you are creating a profile set to be used to create a device-linked profile or to be used to create a concatenated color world. For a list of the rules defining the types of profiles you can use for these purposes, see the `CWNewLinkProfile` function description and the `CWConcatColorWorld` function description in the “ColorSync Manager Reference for Applications and Device Drivers” chapter of the *Advanced Color Imaging Reference*.

3. Identify the CMM to be used for the color processing.

Each of the profiles whose references you give identifies the preferred CMM to be used for color processing involving that profile. To perform color transformation using a series of profiles, the ColorSync Manager uses only one CMM. You use the `CMConcatProfileSet.keyIndex` field to identify the index into the array corresponding to the profile whose preferred CMM is to be used. The array is 0 based, so you must specify the `CMConcatProfileSet.keyIndex` value as a number in the range of 0 to count-minus-1. Count is the number of elements in the array.

4. Call the `CWConcatColorWorld` function to set up the color world.

You pass the `CWConcatColorWorld` function a parameter of type `CMConcatProfileSet` to specify the profile array, and the function returns a color world reference. To perform color matching or gamut checking using the profiles comprising a color world, you call the low-level function passing it the reference to the color world.

Using a device-linked profile for the low-level functions entails additional steps described in “Creating and Using Device-Linked Profiles,” beginning on page 4-53.

Matching Colors Using the Low-Level Functions

Using the low-level `CWMatchPixMap` or `CWMatchBitmap` ColorSync Manager function, your application can match the colors of a pixel image or a bitmap image to the display’s color gamut and display the image without relying on QuickDraw.

Color matching occurs relatively quickly, but for a session involving a large pixel image or bitmap image, the color-matching process may take some time. To keep your user informed, you can provide a progress-reporting function. For example, your function can display an indicator, such as a thermometer, to the user to depict how much of the matching has been done and how much remains. Your function can also allow the user to terminate the color-matching process before it completes.

When your application calls either the `CWMatchPixMap` function or the `CWMatchBitmap` function, you can pass the function a pointer to your callback progress-reporting function and a reference constant containing data, such as the thermometer dialog box’s window reference. When the CMM used to match the colors calls your progress-reporting function, it passes the reference

constant to it. If you provide a progress-reporting function, here is how you should declare the function if you were to name it `MyCMBitmapCallbackProc`:

```
pascal Boolean MyCMBitmapCallbackProc (long progress, void *refCon);
```

For a complete description of the progress-reporting function declaration, see the chapter “ColorSync Manager Reference for Applications and Device Drivers” of the *Advanced Color Imaging Reference*.

To use the `CWMatchPixMap` and `CWMatchBitmap` functions, your application must first set up a color world that specifies the profiles involved in the color-matching session. The color world establishes how matching will take place between the profiles. For information on how to create a color world, see “Creating a Color World for Color Matching and Checking Using the Low-Level Functions” on page 4-27. Listing 4-5 on page 4-32 shows how to match the colors of a pixel map or a bitmap using the ColorSync Manager low-level functions that take a color world.

The ColorSync Manager uses the `PixMap` data type defined by Color QuickDraw. The ColorSync Manager defines and uses the `cmBitmap` data type, based on the classic QuickDraw `Bitmap` data type.

Matching the Colors of a Pixel Map to the Display's Color Gamut

Your application can call the `CWMatchPixMap` function to match the colors of a pixel image to the display's color gamut using a color world that you had previously created. The color world must be based on the source profile for the device used to create the pixel image and the system profile for the system's display.

To match the colors of a pixel image to the display's color gamut, the source profile for the color world must specify a data color space of RGB as its `dataColorSpace` element value to correspond to the pixel map data type, which is implicitly RGB. If the source profile you specify for the color world is the original source profile used to create the pixel image, most likely these values match. However, if you want to verify that the source profile's `dataColorSpace` element specifies RGB, you can use the `CMGetProfileHeader` function to obtain the profile header. The profile header contains the `dataColorSpace` element field. For a pixel image, the display profile's `dataColorSpace` element must also be set to RGB; this is the color space commonly used for displays.

If the source profile is embedded in the document containing the pixel map, your application can extract the profile and open a reference to it before you

create the color world. For information on how to extract an embedded profile, see “Extracting Profiles Embedded in Pictures” on page 4-38. If the source profile is installed in the ColorSync™ Profiles folder, your application can display a list of profiles to the user to allow the user to select the appropriate one.

Listing 4-5 on page 4-32 shows how to set up a color world to be used for color matching of either a pixel map or a bitmap. After setting up the color world, the `MyMatchImage` function calls the `CWMatchPixMap` function to match the pixel map in place.

Matching the Colors of a Bitmap Image to the Display's Color Gamut

Matching the colors of a bitmap image to the current system's display is similar to the process of matching a pixel map's colors except that the data type of a bitmap image is explicitly stated in the `space` field of the bitmap. A bitmap image can be specified using any of the following data types: `cmGrayASpace`, `cmRGB16Space`, `cmRGB32Space`, `cmARGB32Space`, `cmCMYK32Space`, `cmHSV32Space`, `cmHLS32Space`, `cmYXY32Space`, `cmXYZ32Space`, `cmLUV32Space`, or `cmLAB32Space`. The data type of the source bitmap image must correspond to the data color space specified by the color world's source profile.

When you call the `CWMatchBitmap` function, you can pass it a pointer to a bitmap to hold the resulting image. In this case, you must allocate the pixel buffer pointed to by the `image` field of the `CMBitmap` structure. Because the `CWMatchBitmap` function allows you to specify a separate bitmap to hold the resulting color-matched image, you must ensure that the data type you specify in the `space` field of the resulting bitmap matches the destination's color data space.

Rather than creating a bitmap for the color-matched image, you can match the bitmap in place. To do so, you specify `NULL` instead of passing a pointer to a resulting bitmap. In this case, the data color space of the display's system profile must match the data type of the source bitmap image.

The latter portion of the code in Listing 4-5 shows how to set up a bitmap for the resulting color-matched image before calling the `CWMatchBitmap` function to perform the color matching. The `MyMatchImage` function, depicted in this sample listing, uses the profile of the device that produced the image as the source profile and the system profile as the destination profile.

Listing 4-5 Matching the colors of a pixel map or a bitmap using a color world

```

void MyMatchImage(void)
{
    CMError          cmErr;
    CMProfileRef     sourceProf;
    CMProfileRef     sysProf;
    CMWorldRef       cw;
    CMBitmap         bitmap;

    /* set up a color world */

    cmErr = MyGetImageSourceProfile(&sourceProf);

    if (cmErr == noErr)
    {
        cmErr = CMGetSystemProfile(&sysProf);
    }

    if (cmErr == noErr)
    {
        cmErr = NCWNewColorWorld(&cw, sourceProf, sysProf);

        /* close profiles after setting up color world */
        (void) CMCloseProfile(sourceProf);
        (void) CMCloseProfile(sysProf);
    }

    /* match pixmap */

    if (cmErr == noErr)
    {
        cmErr = CWMatchPixMap(cw, gpPixMap, (CMBitmapCallbackUPP) NULL, NULL);
    }

    /* match CMBitmap */

    if (cmErr == noErr)
    {
        /* CMBitmaps corresponding to QD 16 & 32 bit/pixel, RGBDirect
           pixmaps are supported by the ColorSync Manager */
    }
}

```

```

    if ((*gpPixMap).pixelType != RGBDirect)
    {
        cmErr = paramErr;
    }
}

if (cmErr == noErr)
{
    /* set local CMBitmap structure so that it describes the
       pixmap */
    bitmap.image= (*gpPixMap).baseAddr;
    bitmap.width= (*gpPixMap).bounds.right - (*gpPixMap).bounds.left;
    bitmap.height= (*gpPixMap).bounds.bottom - (*gpPixMap).bounds.top;
    /* mask QD rowBytes flag bits */
    bitmap.rowBytes= (*gpPixMap).rowBytes & 0x3ffe;

    switch ((*gpPixMap).pixelSize)
    {
        case 16:
            bitmap.pixelSize= 16;
            bitmap.space= cmRGB16Space;
            break;

        case 32:
            bitmap.pixelSize= 32;
            bitmap.space = cmRGB32Space;
            break;

        default:
            cmErr = paramErr;
            break;
    }

    /* CMBitmap fields not used by the ColorSync Manager */
    bitmap.user1= 0;
    bitmap.user2= 0;
}

if (cmErr == noErr)
{
    /* match in place */

```

Developing ColorSync-Supportive Applications

```

cmErr = CWMatchBitmap(cw, &bitmap, (CMBitmapCallbackUPP) NULL, NULL,
                      (CMBitmap*) NULL);

/* dispose of the colorworld */
CWDisposeColorWorld(cw);
}
}

```

Embedding Profiles in Documents and Pictures

When the user creates and saves a document or picture containing a color image created or modified with your application, your application can provide for future color matching by saving—along with that document or picture—the profile for the device on which the image was created or modified. To get the system profile, your application can use the `CMGetSystemProfile` function described in “Identifying the Current System Profile” on page 4-19 and in the chapter “ColorSync Manager Reference for Applications and Device Drivers” of the *Advanced Color Imaging Reference*. “Searching for Profiles in the ColorSync™ Profiles Folder,” beginning on page 4-49 describes other functions your application can use to search for device profiles.

When embedding source profiles in the documents created by your application, you can store them in any manner that you choose. In the resource fork of the document file, for example, you may choose to have your application store one profile for an entire image, or a separate profile for every object in an image, or a separate profile for every device on which the user modified the image.

When embedding source profiles in PICT file pictures, your application should use the `cmComment` picture comment, which has a `kind` value of 224 and is defined for embedded version 2.0 profiles. This comment is followed by a 4-byte selector that describes the type of data in the comment. The following selectors are currently defined:

Selector	Description
0	Beginning of a version 2.0 profile. Profile data to follow.
1	Continuation of version 2.0 profile data. Profile data to follow.
2	End of version 2.0 profile data. No profile data follows.

Because the `dataSize` parameter of the `PicComment` procedure is a signed 16-bit value, the maximum amount of profile data that can be embedded in a single picture comment is 32,763 bytes (32,767 – 4 bytes for the selector). You can embed a larger profile by using multiple picture comments of selector type 1. You must embed the profile data in consecutive order, and you must conclude the last piece of profile data by embedding a picture comment of selector type 2.

All embedded version 2.0 profiles, including those that fit within a single picture comment, must be followed by the end-of-profile picture comment (selector 2), as shown in the following examples:

Example 1: Embedding a 20K profile.

```
PicComment kind=224, dataSize=20K+4, selector=0, profile data=20K
PicComment kind=224, dataSize=4, selector=2
```

Example 2: Embedding a 50K profile.

```
PicComment kind=224, dataSize=32K, selector=0, profile data=32K-4
PicComment kind=224, dataSize=18K+8, selector=1, profile data=18K+4
PicComment kind=224, dataSize=4, selector=2
```

For version 1.0 of the ColorSync Manager, picture comment types `cmBeginProfile` and `cmEndProfile` are used to begin and end a picture comment. The `cmBeginProfile` comment is not supported for ColorSync Manager version 2.0 profiles; however, the `cmEndProfile` comment can be used to end the current profile and begin using the system profile for both ColorSync 1.0 and 2.0. (Following a `cmEndProfile` comment, the ColorSync Manager reverts to the system profile.) The `cmEnableMatching` and `cmDisableMatching` picture comments are used to begin and end color matching in both ColorSync 1.0 and 2.0. See *Inside Macintosh: Imaging With QuickDraw* for more information about picture comments.

It is important to understand that for embedded profiles to be used properly, the currently effective profile must be correctly terminated by a picture comment of kind `cmEndProfile` after drawing operations using that profile are performed. If you do not specify a picture comment to end the profile, the profile will remain in effect until the next embedded profile is introduced with a picture comment of kind `cmBeginProfile`. It is good practice to always pair use of the `cmBeginProfile` and `cmEndProfile` picture comments. When the ColorSync Manager encounters a `cmEndProfile` picture comment, it restores use of the system profile for matching until it encounters another `cmBeginProfile` picture comment.

The ColorSync Manager provides the `NCMUseProfileComment` function, which automates the process of embedding a profile. This function generates the picture comments required to embed the specified profile into the open picture. It calls the `QuickDraw PicComment` function with a picture comment `kind` value of `cmComment` and a 4-byte selector that describes the type of data in the picture comment: 0 to begin the profile, 1 to continue, and 2 to end the profile. If the size in bytes of the profile and the 4-byte selector together exceed 32 KB, this function segments the profile data and embeds the multiple segments in consecutive order using selector 1 to embed each segment.

However, the `NCMUseProfileComment` function does not terminate the embedded profile with a picture comment to turn off use of the profile.

You are responsible for adding the picture comment of `kind cmEndProfile`. If a picture comment was not specified to end the profile following the drawing operations to which the profile applies, the profile will remain in effect until the next embedded profile is introduced with a picture comment of `kind cmBeginProfile`.

Listing 4-6 shows how to embed a profile in a picture file. The `MyPrependProfileToPicHandle` function depicts how to create a new picture and embed the profile for the device used to create the picture before the picture. The reference for the profile is passed to the `MyPrependProfileToPicHandle` function as the `prof` parameter. Notice that after the `MyPrependProfileToPicHandle` function calls the `NCMUseProfileComment` function to embed the profile, it calls its own `MyEndProfileComment` function to embed a comment of `kind cmEndProfile`, ensuring that the profile is properly terminated.

Listing 4-6 Embedding a profile by prepending it before its associated picture

```
CMError MyPrependProfileToPicHandle (PicHandle pict, PicHandle
                                     *pictNew, CMProfileRef prof)
{
    OSErr          err = noErr;
    CGrafPtr       savePort;
    GDHandle       saveGDev;
    GWorldPtr      smallOff;
    Rect           pictRect;
    CMApplProfileHeader head;
    unsigned long  vers;
```


Developing ColorSync-Supportive Applications

```

unsigned long          flags;

if (prof==nil) return paramErr;

/* get the profile header */
err = CMGetProfileHeader(prof, &head);
if (err) return err;

/* get the version from profile header */
vers = head.cm2.profileVersion;

/* set the embedded profile flag bit */
if (vers >= cmCS2ProfileVersion)
{
    flags = head.cm2.flags;
    head.cm2.flags |= (1<<cmEmbeddedProfile);
    err = CMSetProfileHeader(prof, &head);
    if (err) return err;
}

/* create a temporary graphics world */
err = NewSmallGWorld( &smallOff );
if (err) goto restoreProfFlagAndBail;

GetGWorld( &savePort, &saveGDev );
SetGWorld( smallOff, nil );
pictRect = (**pict).picFrame;
ClipRect( &pictRect );                /* important: set clipRgn */

/* create a new picture */
*pictNew = OpenPicture( &pictRect ); /* start recording */
if (vers == cmCS1ProfileVersion)     /* if version 1 profile... */
    err = paramErr;
else if (vers >= cmCS2ProfileVersion)
    err = NCMUseProfileComment(prof,nil);
DrawPicture( pict, &pictRect )
MyEndProfileComment();
ClosePicture();

```

Developing ColorSync-Supportive Applications

```

    if (err) KillPicture( *pictNew );

    SetGWorld( savePort, saveGDev );
    DisposeGWorld( smallOff );

restoreProfFlagAndBail:
    /* restore the original flag */
    if (vers >= cmCS2ProfileVersion)
    {
        head.cm2.flags = flags;
        err = CMSetProfileHeader(prof, &head);
    }
    return err;
}

```

Here is the application-defined `MyEndProfileComment` function that the code in Listing 4-6 calls to add the `cmEndProfile` picture comment to terminate the profile:

```

void MyEndProfileComment (void)
{
    PicComment( cmEndProfile, 0, 0 );
}

```

Extracting Profiles Embedded in Pictures

To color match or gamut check a picture embedded in a document, your application must first extract the source profile used when the image was created if the profile is embedded in the document along with the picture, and then open a reference to the profile. This process entails locating and identifying the profile for the image within the document and transferring the profile data from the document file.

Note

If you use the high-level `NCMDrawMatchedPicture` function, you do not need to extract the source profile from the PICT file. ♦

To extract an embedded profile, your application can use the `CMUnflattenProfile` function. This function takes a pointer to a low-level data-transfer function that your application must supply to transfer the profile data from the document containing it. This function assumes that your low-level data-transfer function is informed about the context of the profile. After all of the profile data has been transferred, the `CMUnflattenProfile` function returns the file specification for the profile.

When your application calls the `CMUnflattenProfile` function, the ColorSync Manager uses the Component Manager to pass the pointer to your low-level data-transfer function along with the reference constant to the preferred CMM specified by the source profile. If available, the preferred CMM calls your low-level data-transfer function. (If the preferred CMM is not available, the ColorSync Manager follows the CMM selection algorithm described in “How the ColorSync Manager Selects the CMM to Be Used,” beginning on page 4-7, to determine which CMM to use.) The CMM calls your low-level data-transfer function, directing it to open the file containing the profile, read segments of the profile data, and return the data to the CMM’s calling function.

The CMM communicates with your low-level data transfer-function using a command parameter to identify the operation to be performed. To facilitate the transfer of profile data from the file to the CMM, the CMM passes to your function a pointer to a data buffer for data, the size in bytes of the profile data your function should return, and the reference constant passed from the calling application.

On return, your function passes to the CMM segments of the profile data and the number of bytes of profile data you actually return.

The following listings are portions of a sample application called `CSDemo`. You can find the complete sample application in the ColorSync Samples folder of the CD-ROM included with *Advanced Color Imaging on the Mac OS*. These listings assume that all variables beginning with a lowercase letter `g` are global variables previously defined. The application uses global variables to pass data

between functions that do not include reference constant parameters. The listings shown here cover two primary steps:

- “Step 1: Counting the Profiles in the PICT File,” beginning on page 4-40, shows the portion of the application that sets up the port and draws the picture using the bottleneck routines in order to count the number of profiles associated with the picture. The application-defined functions `MyCountProfilesInPicHandle` and `MyCountProfilesCommentProc` perform these processes.
- “Step 2: Extracting the Profile,” beginning on page 4-42, consists of three hierarchical parts that locate a profile, flatten it, and open a reference to the profile. In order to perform these tasks, the code must again draw the picture using the bottleneck routines.

Step 1: Counting the Profiles in the PICT File

Given a `picHandle` value to the picture containing the embedded profile, the sample code shown in Listing 4-7 counts the number of profiles in the picture in order to be able to identify the profile to be extracted.

The `MyCountProfilesInPicHandle` function sets up the port and its bottlenecks and initializes its global counter, which holds a single count summing both ColorSync 1.0 profiles and version 2.0 profiles. The code must draw the picture in order to be able to observe and count the number of profiles. To count the number of profiles as the picture is being drawn, the code uses the `MyCountProfilesCommentProc` bottleneck procedure, but it does not need to use other bottleneck procedures. Therefore, the code defines nonoperational bottleneck routines for the remaining routines. For example, it defines the following function for the `TextProc` bottleneck:

```
static pascal void TextProc (short byteCount, Ptr textAddr,
                             Point numer, Point denom);
```

The code calls its own `MyDrawPicHandleUsingBottleneck` function, not shown here, to draw the picture using the bottleneck routines. Because it must increment the `gCount` global counter for both ColorSync 1.0 profiles and version 2.0 profiles, `MyCountProfilesCommentProc` checks for both types of profiles.

Listing 4-7 Counting the number of profiles in a picture

```

CMError MyCountProfilesInPicHandle (PicHandle pict, unsigned long *count)
{
    OSErr      err = noErr;
    CQDProcs   procs;

    /* set up bottleneck for picComments so we can count the profiles */
    SetStdCProcs(&procs);
    procs.textProc   = NewQDTextProc(TextProc);
    procs.lineProc   = NewQDLineProc (LineProc);
    procs.rectProc   = NewQDRectProc (RectProc);
    procs.rRectProc  = NewQDOvalPro (RRectProc);
    procs.ovalProc   = NewQDOvalProc (OvalProc);
    procs.arcProc    = NewQDArcProc (ArcProc);
    procs.polyProc   = NewQDPolyProc (PolyProc);
    procs.rgnProc    = NewQDRgnProc (RgnProc);
    procs.bitsProc   = NewQDBitsProc (BitsProc);
    procs.commentProc = NewQDCommentProc(MyCountProfilesCommentProc);
    procs.txMeasProc = NewQDTxMeasProc (TxMeasProc);

    /* initialize the global counter to be incremented by the commentProc*/
    gCount = 0;

    /* draw the picture in order to count the profiles */
    err = MyDrawPicHandleUsingBottlenecks (pict, procs, nil);

    /* obtain the result from the count global variable */
    *count = gCount;

    /* clean up and return*/
    DisposeRoutineDescriptor(procs.textProc);
    DisposeRoutineDescriptor(proc.lineProc);
    DisposeRoutineDescriptor(procs.rectProc);
    DisposeRoutineDescriptor(procs.rRectProc);
    DisposeRoutineDescriptor(procs.ovalProc);
    DisposeRoutineDescriptor(procs.arcProc);
    DisposeRoutineDescriptor(procs.polyProc);
    DisposeRoutineDescriptor(procs.rgnProc);
    DisposeRoutineDescriptor(procs.bitsProc);
    DisposeRoutineDescriptor(procs.commentProc);
}

```

```

        DisposeRoutineDescriptor(procs.txMeasProc);
    }

pascal void MyCountProfilesCommentProc (short kind, short dataSize, Handle
                                         dataHandle)
{
    long    selector;

    switch (kind)
    {
        case cmBeginProfile
            gCount ++;                /* we found a ColorSync 1.0 profile */
                                     /* increment the counter*/
            break;

        case cmComment;
            if (dataSize <= 4) break; /* dataSize is too small for selector
                                     so break and get the selector
                                     from the first long */
            selector = *((long *)(*dataHandle));
            if (selector == cmBeginProfileSel)
                gCount ++;            /* we found a version 2 profile; increment
                                     the counter */
            break;
    }
}

```

Step 2: Extracting the Profile

This part of the sample application identifies the profile to be flattened, flattens the profile, and creates a temporary profile disposing of the original one.

Part A: Calling the Unflatten Function

Listing 4-8 shows the `MyGetIndexedProfileFromPichandle` entry point function that drives the process of unflattening the profile. The code creates a universal procedure pointer (UPP), `MyflattenUPP`, that points to the low-level data-transfer procedure.

A PICT handle may contain more than one profile. To identify the profile to be unflattened, the `MyGetIndexedProfileFromPichandle` function contains an `index` parameter that passes in the profile's index. The code stores the index in the

global variable `gIndex` so that the value is accessible by the application's other functions that check for the correct profile and extract it. Then, the code calls the `CMUnflattenProfile` function, passing it the `MyflattenUPP` pointer. This invokes the `MyUnflattenProc` function shown in Listing 4-9.

When the `CMUnflattenProfile` function returns, the code calls the `CMOpenProfile` function to open a reference to the file-based profile; then it calls `CMCopyProfile` to create a temporary profile. Finally, the code disposes of the original profile. The code creates the temporary profile and disposes of the original in order to adhere to the copyright protection for embedded profiles set through the profile header `flags` field setting.

Listing 4-8 Calling the `CMUnflattenProfile` function to extract an embedded profile

```
CMError MyGetIndexedProfileFromPicHandle (PicHandle pict, unsigned long index,
                                         CMProfileRef *prof, CMProfileLocation
                                         *profLoc)
{
    unsigned long          refCon;
    CMFlattenUPP           MyflattenUPP;
    CMError                cmErr = noErr;
    Boolean                preferredCMMNotFound;
    FSSpec                 tempSpec;
    CMProfileRef           tempProf;
    CMProfileLocation       tempProfLoc;

    MyflattenUPP = NewCMFlattenProc(MyUnflattenProc); /* create a universal
                                                         procedure pointer for unflatten
                                                         procedure */

    /* assumes that index <= count */

    refCon = (unsigned long) pict;
    gIndex = index;

    cmErr = CMUnflattenProfile(&tempSpec, MyflattenUPP, (void*)&refCon,
                              &preferredCMMNotFound); /* this function invokes the
                                                         MyUnflattenProc shown in Listing 4-9 */
    DisposeRoutineDescriptor(MyflattenUPP);
```

```

if (cmerr) return cmerr;

tempProfLoc.locType = cmFileBasedProfile;
tempProfLoc.u.fileLoc.spec = tempSpec;

cmErr = CMOpenProfile(&tempProf, &tempProfLoc);
if (cmerr) return cmerr;

cmErr = CMCopyProfile(prof, profLoc, tempProf);
cmErr = CMCloseProfile(tempProf);
cmErr = FSpDelete(&tempSpec);

return cmerr;
}

```

B: Calling the Unflatten Function

When the code in Listing 4-8 calls the `CMUnflattenProc` function passing it a pointer to the `MyUnflattenProc` function, the `MyUnflattenProc` function shown in Listing 4-9 gets called by the CMM to perform the low-level profile data transfer from the document file.

When the CMM calls this function with an open command, the function initializes global variables, creates a graphics world, and installs the bottleneck procedures in the graphics world. The only bottleneck procedure that is actually used is `MyUnflattenProfilesCommentProc`, which checks the picture comments as the picture is drawn offscreen in order to identify the desired profile.

When the CMM calls the `MyUnflattenProc` function with a read command, the function reads the appropriate segment of data from a chunk and returns it. To accomplish this, the code calls the `MyDrawMatchedPicture` function with the appropriate bottleneck procedure installed. In turn, this invokes the `MyUnflattenProfilesCommentProc` shown in Listing 4-10.

When the CMM calls the code with a close command, the code releases the memory it used and disposes of the graphics world and bottlenecks.

Listing 4-9 The unflatten procedure

```

pascal OSErr MyUnflattenProc (long command, long *sizePtr, void
                             *dataPtr, void *refConPtr)
{
    OSErr          err = noErr;
    static CQDProcs procs;
    static GWorldPtr offscreen;
    PicHandle      pict;

    switch (command)
    {
        case cmOpenReadSpool:
            err = NewSmallGWorld(&offscreen);
            if (err) return err;

            SetStdCProcs(&procs);
            procs.textProc    = NewQDTextProc (MyNoOpTextProc);
            procs.lineProc    = NewQDLineProc (MyNoOpLineProc);
            procs.rectProc    = NewQDRectProc (MyNoOpRectProc);
            procs.rRectProc   = NewQDRRectPro (MyNoOpRRectProc);
            procs.ovalProc    = NewQDOvalProc (MyNoOpOvalProc);
            procs.arcProc     = NewQDArcProc  (MyNoOpArcProc);
            procs.polyProc    = NewQDPolyProc (MyNoOpPolyProc);
            procs.rgnProc     = NewQDRgnProc  (MyNoOpRgnProc);
            procs.bitsProc    = NewQDBitsProc (MyNoOpBitsProc);
            procs.commentProc = NewQDCommentProc (MyUnflattenProfilesCommentProc);
            procs.txMeasProc   = NewQDTxMeasProc (MyNoOpTxMeasProc);

            gChunkBaseHndl = nil;
            gChunkIndex = 0;
            gChunkOffset = 0;
            gChunkSize = 0;
            break;

        case cmReadSpool:
            if (gChunkOffset > gChunkSize)      /* if we overread the last chunk */
            {
                return ioErr;
            }
            if (gChunkOffset == gChunkSize)     /* if we used up the last chunk */

```

Developing ColorSync-Supportive Applications

```

{
    if (gChunkBaseHndl != nil)
    {
        HUnlock(gChunkBaseHndl);    /* dispose of the previous chunk */
        DisposeHandle(gChunkBaseHndl);
        gChunkBaseHndl = nil;
    }
    gChunkIndex++;                /* read in a new chunk */
    gChunkOffset = 0;
    gCount = 0;
    gChunkCount = 0;
    pict = *((PicHandle *)refConPtr);
    err = MyDrawPicHandleUsingBottlenecks (pict, procs, offscreen);
        /* this invokes MyUnflattenProfilesCommentProc shown in
           Listing 4-10 */
    if (gChunkBaseHndl==nil)      /* check to see if we're overread */
        return ioErr;
    HLock(gChunkBaseHndl);
}
if (gChunkOffset < gChunkSize)
{
    *sizePtr = MIN(gChunkSize-gChunkOffset, *sizePtr);
    BlockMove((Ptr)((*gChunkBaseHndl)[gChunkOffset])),
        (Ptr)dataPtr, *sizePtr );
    gChunkOffset += (*sizePtr);
}
break;

case cmCloseSpool:
    if (gChunkBaseHndl != nil)
    {
        HUnlock(gChunkBaseHndl);    /* dispose of the previous chunk */
        DisposeHandle(gChunkBaseHndl);
        gChunkBaseHndl = nil;
    }
    DisposeGWorld(offscreen);
    DisposeRoutineDescriptor(procs.MyNoOpTextPrc);
    DisposeRoutineDescriptor(procs.MyNoOpLinePrc);
    DisposeRoutineDescriptor(procs.MyNoOpRectPrc);
    DisposeRoutineDescriptor(procs.MyNoOpRRectPrc);
    DisposeRoutineDescriptor(procs.MyNoOpOvalPrc);
}

```

Developing ColorSync-Supportive Applications

```

DisposeRoutineDescriptor(procs.MyNoOpArcProc);
DisposeRoutineDescriptor(procs.MyNoOpPolyPrc);
DisposeRoutineDescriptor(procs.MyNoOpRgnProc);
DisposeRoutineDescriptor(procs.MyNoOpBitsProc);
DisposeRoutineDescriptor(procs.MyUnflattenProfilesCommentPrc);
DisposeRoutineDescriptor(procs.MyNoOpTxMeasPrc);
break;

default:
    break;
}
return err;
}

```

Part C: Calling the Comment Procedure

When the `MyUnflattenProc` function's `MyDrawPicHandleUsingBottlenecks` function calls the `MyUnflattenProfilesCommentProc` function, the function shown in Listing 4-10 finds the profile identified by the index, finds the correct segment of data within the profile, and stores the data in the `gChunkBaseHndl` global variable.

Listing 4-10 The comment procedure

```

pascal void MyUnflattenProfilesCommentProc (short kind, short
                                           dataSize, Handle dataHandle)
{
    long    selector;
    OSErr   err;

    if (gChunkBaseHndl != nil) return;
        /* the handle is in use; this shouldn't happen */
    if (gCount > gIndex) return;
        /* we have already found the profile */

    switch (kind)
    {
    case cmBeginProfile:
        gCount ++;          /* we found a version 1 profile */
        gChunkCount = 1;    /* v1 profiles should only have 1 chunk */
    }
}

```

Developing ColorSync-Supportive Applications

```

if (gCount != gIndex) break;
        /* this is not the profile we're looking for */
if (gChunkCount != gChunkIndex) break;
        /* this is not the chunk we're looking for */
gChunkBaseHndl = dataHandle;
err = HandToHand(&gChunkBaseHndl);
gChunkSize = dataSize;
gChunkOffset = 0;
break;

case cmComment:
    if (dataSize <= 4 ) break;
        /* the dataSize too small for selector, so break */
    selector = *((long *)(&dataHandle));
        /* get the selector from the first long in data */
    switch (selector)
    {
        case cmBeginProfileSel:
            gCount ++;
                /* we found a version 2 profile */
            gChunkCount = 1;
            if (gCount != gIndex) break;
                /* this is not the profile we're looking for */
            if (gChunkCount != gChunkIndex) break;
                /* this is not the chunk we're looking for */
            gChunkBaseHndl = dataHandle;
            err = HandToHand(&gChunkBaseHndl);
            gChunkSize = dataSize;
            gChunkOffset = 4;
            break;

        case cmContinueProfileSel:
            gChunkCount ++;
            if (gCount != gIndex) break;
                /* this is not the profile we're looking for */
            if (gChunkCount != gChunkIndex) break;
                /* this is not the chunk we're looking for */
            gChunkBaseHndl = dataHandle;
            err = HandToHand(&gChunkBaseHndl);
            gChunkSize = dataSize;
            gChunkOffset = 4;
            break;
    }
}

```

```

case cmEndProfileSel:
    /* check to see if we're overreading */
    gChunkCount = 0;
    break;
}
break;
}
}

```

Searching for Profiles in the ColorSync™ Profiles Folder

Your application can use the ColorSync Manager search functions to obtain a list identifying profiles in the ColorSync™ Profiles folder that meet specifications you supply in a search record. For example, you can use these functions to find all profiles for printers that meet certain criteria defined in the profile. Your application can walk through the result listing that identifies these profiles and obtain the name and script code of each profile corresponding to a specific index in the list. Your application can then display a selection menu showing the names of the profiles to your user. Listing 4-11 shows sample code that takes an approach similar to one this example describes.

Listing 4-11 defines values for the search specification record fields, including the search mask, and assigns those values to the record's fields after initializing the search result.

Then the code calls the `CMNewProfileSearch` function to search the ColorSync™ Profiles folder for profiles that meet the search specification requirements. The function returns a one-based count of the profiles matching the search specification and a reference to the search result list of the matching profiles.

Next the function calls the `CMSearchGetIndProfile` function to obtain a reference to a specific profile corresponding to a specific index into the search result list. Passing the profile reference returned by the `CMSearchGetIndProfile` function as the `foundProf` parameter, the code calls the `CMGetScriptProfileDescription` function to obtain the profile name and script code.

Finally, the code cleans up, calling the `CMCloseProfile` function to close the profile and the `CMDisposeProfileSearch` function to dispose of the search result list.

Listing 4-11 Searching for specific profiles in the ColorSync™ Profiles folder

```

/* field definitions for search */
#define kCMMType      'appl'          /* ColorSync default CMM */
#define kProfileClass cmDisplayClass /* monitor */
#define kAttr0        0x00000000
#define kAttr1        0x00000002     /* Macintosh standard gamma */

#define kSearchMask    (cmMatchProfileCMMTypecmMatchProfileClasscmMatchAttributes)

void MyProfileSearch(void)
{
    CMError          cmErr;
    CMProfileRef      foundProf;
    Str255            profName;
    ScriptCode        profScript;
    CMSearchRecord     searchSpec;
    CMProfileSearchRef searchResult;
    unsigned long      searchCount;
    unsigned long      i;

    /* init for error handling */

    searchResult = NULL;

    /* specify search */
    searchSpec.CMMType= kCMMType;
    searchSpec.profileClass= kProfileClass;
    searchSpec.deviceAttributes[0]= kAttr0;
    searchSpec.deviceAttributes[1]= kAttr1;

    searchSpec.searchMask = kSearchMask;

    searchSpec.filter= NULL;          /* filter proc is not used */

    cmErr = CMNewProfileSearch(&searchSpec, NULL, &searchCount, &searchResult);

    if (cmErr == noErr)
    {
        for (i = 1; i <= searchCount; i++)
        {

```

```

    if (CMSearchGetIndProfile(searchResult, i, &foundProf) != noErr)
    {
        break;
    }

    cmErr = CMGetScriptProfileDescription(foundProf, profName, &profScript);

    if (cmErr == noErr)
    {
        /* assume profile name ScriptCode is smRoman */
        (void) printf("%s\n", p2cstr(profName));
    }

    (void) CMCloseProfile(foundProf);
}

if (searchResult != NULL)
{
    CMDisposeProfileSearch(searchResult);
}
}

```

Checking Colors Against a Destination Device's Gamut

Different imaging devices (scanners, displays, printers) work in different color spaces, and each can have a different gamut or range of colors that they can produce. The process of matching colors between devices entails adjusting the colors of an image from the color gamut of one device to the color gamut of another device so that the resulting image looks as similar as possible to the original image. Not all colors can be rendered on all devices. The rendering intent used in the color transformation process dictates how the colors are matched, strongly influencing the outcome. Your application can give your user some control over the outcome by allowing the user to select the rendering intent to be used. However, some users might want to know in advance which colors are out of gamut for the destination device so that they can choose other appropriate colors within the gamut.

Using the ColorSync Manager low-level color-checking functions, your application can check the colors of a pixel map (using the `CWCheckPixMap` function), the colors of a bitmap (using the `CWCheckBitmap` function), or a list of

colors (using the `CWCheckColors` function) against the color gamut of the destination device and warn your user when a color she or he chooses is out of gamut for that device.

There are a number of ways in which your application can provide gamut-checking services. For example, you can use gamut checking to see if a given color is reproducible on a particular printer. If the color is not directly reproducible—that is, if it is out of gamut—you could alert the user to that fact.

You can allow your user to specify a list of colors that fall within the gamut of a source device to see if they fit within the gamut of a destination device before the user color matches an image. Your application could display the results in a window, indicating which colors are in the gamut and which are out. This feature, too, gives the user the opportunity to test colors and select different ones for portions of an image whose colors fall out of gamut. To handle this feature, your application can call the `CWCheckColors` function.

In addition to providing features that allow your user to anticipate which colors are out of gamut for a particular device, your application can also show results. Your application can provide a print preview dialog box, showing which colors in a printed image, for example, are out of gamut for the image as it appears on the screen.

For an image that your application prepares, for example, your application can present a print preview dialog box that signifies those colors within the image that the printer cannot accurately reproduce. Your application can also allow users to choose whether and how to match colors in the image with those available on the printer.

You can provide a gamut-checking feature that marks the areas of a displayed image, showing the colors that do not fall within the destination device's gamut. For example, your application can color check an image against a destination device and create a black-and-white version of the image drawn to the display using black to indicate the portions of the source image that are out of gamut. The sample application called `CSDemo` that is provided in the ColorSync Samples folder of the CD-ROM included with this book takes this approach.

Creating and Using Device-Linked Profiles

To accommodate users who use a specific configuration requiring a combination of device profiles and possibly nondevice profiles repeatedly over time, your application can create device-linked profiles. A device-linked profile offers your user a means of saving and storing a series of profiles corresponding to a specific configuration in a concatenated format. This feature provides an economy of effort for both your application and its user.

There are many uses for device-linked profiles. For example, a user might want to store multiple profiles, such as various device profiles and color space profiles associated with the creation and editing of an image.

Most users use the same device configuration to scan, view, and print graphics over a period of time, often soft proofing images before they print them. To enhance your application's soft-proofing feature, you can allow your user to store the contents of the profiles involved in the soft-proofing process in a device-linked profile. Each time a user enacts your application's soft-proofing feature, your application can use the appropriate device-linked profile for the configuration instead of opening profile references to each of the profiles in order to create a color world to pass to the color-matching functions. (For information about soft proofing, see "Providing Soft Proofs" on page 4-56.)

A device-linked profile is especially useful when a scanner application does not embed the source profile in the document containing the image it creates. By storing the scanner's profile, your application eliminates the need to query the user for the appropriate source profile each time the user wants to soft proof using the configuration involving that scanner.

A user may want to see how a scanned image will look when printed using a specific printer. The user may want to look at many images captured on the same scanner at different times before printing the image. Because the same devices are involved in the process, if your application has offered the user the opportunity to create device-linked profiles, your application could display a list of device-linked profiles that the user had previously created for various configurations and allow the user to select the appropriate one for the current soft-proofing.

Here are the steps your application should take in creating a device-linked profile:

- 1. Open the profiles corresponding to the devices and transformations involved in the configuration and obtain references to them.**

To create a device-linked profile, your application must first obtain references to the profiles involved in the configuration. If the profile for an input device, such as a scanner, is embedded in the document containing the image, you must first extract the profile. For a description of how to obtain a profile reference, see “Obtaining Profile References” on page 4-16. For information describing how to extract a profile from a document, see “Extracting Profiles Embedded in Pictures,” beginning on page 4-38.

- 2. Create an array containing references to the profiles, specifying the profile references in processing order.**

You supply the profile references as an array of type `CMProfileRef` within a data structure of type `CMConcatProfileSet`. The order of the profiles must correspond to the order in which you want the colors of the image to be processed. For example, for soft proofing an image, you should specify the scanner profile reference first, followed by the printer profile reference, and then the display profile reference because the goal is to match the colors of the scanned image to the color gamut of the printer for which the image is destined and then display the results to the user.

In the `count` field, specify a 0-based number identifying how many profiles the array holds. A device-linked profile represents a one-way link between devices.

Here is the `CMConcatProfileSet` data type:

```
struct CMConcatProfileSet {
    unsigned short    keyIndex;
    unsigned short    count;
    CMProfileRef      profileSet[1];
};
```

You must adhere to the rules that govern the type of profiles you can specify in the array. For example, the first and last profiles must be device profiles. For a list of these rules, see the chapter “ColorSync Manager Reference for Applications and Device Drivers” of the *Advanced Color Imaging Reference*.

3. Specify the index corresponding to the profile whose preferred CMM is to be used to perform the processing.

The header of each profile specifies the preferred CMM for that profile. Only one CMM is used for all transformations across the profiles of a device-linked profile, and you must identify the profile whose CMM is to be used by giving the index of that profile in the `keyIndex` field of the `CMConcatProfileSet` data type.

4. Using the `CMProfileLocation` data type, provide a file specification for the new device-linked profile.

If the `CWNewLinkProfile` function completes successfully, the ColorSync Manager creates a device-linked profile in the location that you specify, opens a reference to the profile, and returns the profile reference to your application. To tell the ColorSync Manager where to create the new profile, your application must provide a file specification. The ColorSync Manager defines a data structure of type `CMProfileLocation` containing a `CMProfLoc` union that you use to give a file specification. (You can look at Listing 4-1 on page 4-18, which assigns values to a `CMProfileLocation` data structure.)

5. Call the `CWNewLinkProfile` function to create the device-linked profile.

After you set up `CMConcatProfileSet` and `CMProfileLocation`, your application can call the `CWNewLinkProfile` function, passing these values to it. If the function completes successfully, it returns a reference to the newly created device-linked profile.

Note that you cannot embed a device-linked profile into a document along with an image that uses it.

6. Using the `CWConcatColorWorld` function, create a color world based on the device-linked profile.

You can use a device-linked profile with the low-level ColorSync Manager functions only. To use a device-linked profile for a color-matching or color gamut-checking function, you must first create a color world using the `CWConcatColorWorld` function, passing to it a data structure of type `CMConcatProfileSet`. The `CMConcatProfileSet` data structure is the same data type that you used to specify the array of profiles when you created the new device-linked profile. To create the color world, however, you specify the device-linked profile as the only member of the `CMConcatProfileSet` array. If the `CWConcatColorWorld` function completes successfully, it returns a reference to a color world that your application can pass to other low-level functions for color-matching and color-checking sessions. A device-linked

profile remains intact and available for use again after your application calls the `CWDisposeColorWorld` function to dispose of the concatenated color world.

Considerations

Here are some points to consider about how the ColorSync Manager uses information contained in the profiles comprising a device-linked profile:

- When you use a device-linked profile, the quality flag setting—indicating normal mode, draft mode, or best mode—specified by the first profile prevails for the entire session; the quality flags of following profiles in the sequence are ignored. The quality flag setting is stored in the `flags` field of the profile header.
- The rendering intent specified by the first profile is used to color match to the second profile, the rendering intent specified by the second profile is used to color match to the third profile, and so on through the series of concatenated profiles.

When your application is finished with the device-linked profile, it must close the profile with the `CMCloseProfile` function.

Providing Soft Proofs

Using the ColorSync Manager, your application can provide your users with a soft-proofing feature to enable them to preview the printed results of a color image on the system's display or local printer without actually outputting the image to the printer that will produce the final image. The destination printer's profile provides the ColorSync Manager with the information required to determine how the colors of the image will appear when printed. You can soft proof an image by showing on the system's display the outcome a printer would produce because most displays support a wider color gamut than do printers. Therefore, a display will probably be able to show all the colors a printer could support.

Providing a feature that simulates the printed outcome for the user to preview can save users considerable time and cost by allowing them to intervene and adjust colors before sending the image to a printing shop. For example, without the ability to soft proof and correct the colors of an image using a color management system such as the ColorSync Manager, a graphics designer producing a poster to be printed by a printing press would require the services of a prepress shop to achieve the correct results before sending the image to the

printing press. The graphics designer might print the image to a local desktop printer with a color gamut more limited than that of a printing press and then submit the output to the prepress in order to correct the colors, repeating this process until the results were satisfactory. Your application can eliminate the need for the intermediate steps by allowing the user to color match the image to the color gamut of the final printing press, display the image, and adjust the colors accordingly.

You can use the low-level ColorSync Manager color-matching functions `CWMatchPixMap` and `CWMatchBitmap` to perform the color matching, or you can match a list of colors using the `CWMatchColors` function. To use these functions, your application must first define a color world that encompasses the profiles for the devices involved in the soft-proofing process.

For example, suppose your user intends to create a color image by drawing to the display, then color matching the image to the color gamut of the printing press and printing the image to a local desktop printer before delivering it to the printing press. The user intends to repeat this process until he or she is satisfied with the color rendering. To allow the user to do this, your application must build a color world using the system profile for the display device, the profile for the printing press, and the profile for the local desktop printer; you must specify the profiles in processing order. Because the process involves three profiles, your application must use the `CWConcatColorWorld` function to set up the color world. “Creating a Color World for Color Matching and Checking Using the Low-Level Functions” on page 4-27 describes how to set up a color world.

If your application provides a feature that allows the user to store a series of profiles in a device-linked profile, you can preserve the profiles used for the soft-proofing process for future use by creating a device-linked profile representing the configuration and pass the device-linked profile to the `CWConcatColorWorld` function to set up the color world. For information describing how to create and use a device-linked profile to build a color world, see “Creating and Using Device-Linked Profiles” on page 4-53.

Your application can also use the high-level QuickDraw-based `NCMBeginMatching` and `CMEndMatching` functions for soft proofing of a color image drawn to the display that your user wants to color match to the gamut of a printing press and print to a desktop printer.

The `NCMBeginMatching` function matches the colors using the two profiles that you specify, and the `CMEndMatching` function terminates the color-matching session. Because the `NCMBeginMatching` function takes two profiles only—a

source profile and a destination profile—you must call sets of these functions to enact soft proofing.

It is important to recognize that QuickDraw matches to the most recently added profiles first. Therefore, to use the `NCMBeginMatching` and `CMEndMatching` pair to perform soft proofing from a displayed image to a printing press output image to a desktop printer image, you would first call the `NCMBeginMatching` function with the printing press to desktop printer profile references and then call `NCMBeginMatching` with the display to printing press profile references. QuickDraw will color match all drawing from display to printing press and then to the desktop printer.

To use the `NCMBeginMatching` function, you specify the source and destination profiles to be used. Passing `NULL` as the source profile assures that the ColorSync Manager uses the system profile as the source profile. Similarly, passing `NULL` as the destination profile uses the system profile as the destination profile.

Calibrating a Device

A calibration application either creates a profile or tunes a profile to represent the current state of the device.

A profile contains two types of device information: the actual calibration information describing how to perform the color match and the device settings at the time the match was made, for example, paper type, ink flow, or film exposure time. A device may have several profiles, each for a different setting, such as paper type or ink.

Your calibration program should first turn off matching on the device and generate its image. You should then perform the calibration and generate a profile.

Summary of the ColorSync Manager

Constants

```

/* constants for profile location type */
enum {
    cmNoProfileBase = 0,
    cmFileBasedProfile= 1,
    cmHandleBasedProfile= 2,
    cmPtrBasedProfile= 3
};

/* profile classes */
enum {
    cmInputClass      = 'scnr',
    cmDisplayClass    = 'mntr',
    cmOutputClass     = 'prtr',
    cmLinkClass       = 'link',
    cmAbstractClass   = 'abst',
    cmColorSpaceClass = 'spac'
};

/* signature of the Apple-supplied color management module (CMM) */
enum {
    kDefaultCMMSignature = 'appl'
};

/* commands for calling the application-supplied MyColorSyncDataTransfer */
enum {
    openReadSpool = 1,
    openWriteSpool,
    readSpool,
    writeSpool,
    closeSpool
};

```

Developing ColorSync-Supportive Applications

```

/* picture comment IDs for profiles and color matching */
enum {
    cmBeginProfile      = 220,
    cmEndProfile        = 221,
    cmEnableMatching    = 222,
    cmDisableMatching   = 223,
    cmComment           = 224
};

/* picture comment selectors for the cmComment ID */
enum {
    cmBeginProfileSel    = 0,
    cmContinueProfileSel = 1,
    cmEndProfileSel      = 2
};

/* color space signatures */
enum {
    cmXYZData            = 'XYZ ',
    cmLabData            = 'Lab ',
    cmLuvData            = 'Luv ',
    cmYxyData            = 'Yxy ',
    cmRGBData            = 'RGB ',
    cmGrayData           = 'GRAY',
    cmHSVData            = 'HSV ',
    cmHLSData            = 'HLS ',
    cmCMYKData           = 'CMYK',
    cmCMYData            = 'CMY ',
    cmMCH5Data           = 'MCH5',
    cmMCH6Data           = 'MCH6',
    cmMCH7Data           = 'MCH7',
    cmMCH8Data           = 'MCH8'
};

/* color packing for color spaces */
enum {
    cmNoColorPacking     = 0x0000,
    cmAlphaSpace         = 0x0080,
    cmWord5ColorPacking  = 0x0500,
    cmLong8ColorPacking  = 0x0800,
    cmLong10ColorPacking = 0x0a00,
};

```


Developing ColorSync-Supportive Applications

```

    cmAlphaFirstPacking      = 0x1000,
    cmOneBitDirectPacking    = 0x0b00
};

/* color spaces */
enum {
    cmNoSpace                = 0,
    cmRGBSpace,
    cmCMYKSpace,
    cmHSVSpace,
    cmHLSpace,
    cmYXYSpace,
    cmXYZSpace,
    cmLUVSpace,
    cmLABSpace,
    cmReservedSpace1,
    cmGraySpace,
    cmReservedSpace2,
    cmGamutResultSpace,
    cmRGBASpace              = cmRGBSpace + cmAlphaSpace,
    cmGrayASpace             = cmGraySpace + cmAlphaSpace,
    cmRGB16Space             = cmWord5ColorPacking + cmRGBSpace,
    cmRGB32Space             = cmLong8ColorPacking + cmRGBSpace,
    cmARGB32Space            = cmLong8ColorPacking +
        cmAlphaFirstPacking + cmRGBASpace,
    cmCMYK32Space            = cmLong8ColorPacking + cmCMYKSpace,
    cmHSV32Space             = cmLong10ColorPacking + cmHSVSpace,
    cmHLS32Space             = cmLong10ColorPacking + cmHLSpace,
    cmYXY32Space            = cmLong10ColorPacking + cmYXYSpace,
    cmXYZ32Space            = cmLong10ColorPacking + cmXYZSpace,
    cmLUV32Space            = cmLong10ColorPacking + cmLUVSpace,
    cmLAB32Space            = cmLong10ColorPacking + cmLABSpace,
    cmGamutResult1Space      = cmOneBitDirectPacking +
        cmGamutResultSpace
};

/* rendering intent values for version 2.0 profiles */
enum {
    cmPerceptual              = 0,
    cmRelativeColorimetric    = 1,

```

```

    cmSaturation          = 2,
    cmAbsoluteColorimetric = 3
};

/* PrGeneral operation codes */
enum {
    enableColorMatchingOp= 12,
    registerProfileOp= 13
};

/* color conversion component version */
enum {
    CMConversionInterfaceVersion = 1
};

/* ColorSync Manager element tags and their signatures for version 1.0 profiles */
enum {
    cmCS1ChromTag      = 'chrn',
    cmCS1TRCTag        = 'trc ',
    cmCS1NameTag        = 'name',
    cmCS1CustTag        = 'cust'
};

/* defines for the CMSearchRecord.searchMask field */
enum {
    cmMatchAnyProfile= 0x00000000,
    cmMatchProfileCMMType= 0x00000001,
    cmMatchProfileClass= 0x00000002,
    cmMatchDataColorSpace= 0x00000004,
    cmMatchProfileConnectionSpace = 0x00000008,
    cmMatchManufacturer= 0x00000010,
    cmMatchModel      = 0x00000020,
    cmMatchAttributes= 0x00000040,
    cmMatchProfileFlags= 0x00000080
};

```

Data Structures

```

/* profile location union */
union CMProfLoc {
    CMFileLocation      fileLoc;
    CMHandleLocation    handleLoc;
    CMPtrLocation       ptrLoc;
};

/* profile location structure */
struct CMProfileLocation{
    short      locType;
    CMProfLoc  u;
};

/* file specification for file-based profiles */
struct CMFileLocation {
    FSSpec  spec;
};

/* handle specification for memory-based profiles */
struct CMHandleLocation {
    Handle  h;
};

/* pointer specification for memory-based profiles */
struct CMPtrLocation {
    Ptr  p;
};

/* Apple profile header */
union CMAppleProfileHeader {
    CMHeader      cm1;
    CM2Header     cm2;
};

/* ColorSync Manager profile 2.0 header structure */
struct CM2Header {
    unsigned long      size;
    OSType             CMMType;
    unsigned long      profileVersion;
    OSType             profileClass;
    OSType             dataColorSpace;
};

```

Developing ColorSync-Supportive Applications

```

    OSType                profileConnectionSpace;
    CMDateTime            dateTime;
    OSType                CS2profileSignature;
    OSType                platform;
    unsigned long         flags;
    OSType                deviceManufacturer;
    unsigned long         deviceModel;
    unsigned long         deviceAttributes[2];
    unsigned long         renderingIntent;
    CMFixedXYZColor       white;
    char                  reserved[48];
};

/* concatenated profile set structure */
struct CMConcatProfileSet {
    unsigned short        keyIndex;
    unsigned short        count;
    CMProfileRef          profileSet[1];
};

/* color world information record */
struct CMCWInfoRecord {
    unsigned long         cmmCount;
    CMMInfoRecord         cmmInfo[2];
};

/* color management module (CMM) information record structure */
struct CMMInfoRecord {
    OSType                CMMType;
    long                  CMMVersion;
};

/* profile search record */
struct CMSearchRecord {
    OSType                CMMType;
    OSType                profileClass;
    OSType                dataColorSpace;
    OSType                profileConnectionSpace;
    unsigned long         deviceManufacturer;
    unsigned long         deviceModel;
    unsigned long         deviceAttributes[2];
    unsigned long         profileFlags;
};

```

```

        unsigned long          searchMask;
        CMProfileFilterUPP      filter;
};

/* XYZ color-component values */
typedef unsigned short CMXYZComponent;

/* XYZ color value */
struct CMXYZColor {
    CMXYZComponent    X;
    CMXYZComponent    Y;
    CMXYZComponent    Z;
};

/* fixed XYZ color value */
struct CMFixedXYZColor {
    Fixed            X;
    Fixed            Y;
    Fixed            Z;
};

/* L*a*b* color value */
struct CMLabColor {
    unsigned short    L;
    unsigned short    a;
    unsigned short    b;
};

/* L*u*v* color value */
struct CMLuvColor {
    unsigned short    L;
    unsigned short    u;
    unsigned short    v;
};

/* Yxy color value */
struct CMYxyColor {
    unsigned short    capY;    /* 0..65535 maps to 0..1 */
    unsigned short    x;      /* 0..65535 maps to 0..1 */
    unsigned short    y;      /* 0..65535 maps to 0..1 */
};

```

```

/* RGB color value */
struct CMRGBColor {
    unsigned short    red;
    unsigned short    green;
    unsigned short    blue;
};

/* HLS color value */
struct CMHLSColor {
    unsigned short    hue;
    unsigned short    lightness;
    unsigned short    saturation;
};

/* HSV color value */
typedef struct CMHSVColor {
    unsigned short    hue;
    unsigned short    saturation;
    unsigned short    value;
};

/* CMYK color value */
struct CMCMYKColor {
    unsigned short    cyan;
    unsigned short    magenta;
    unsigned short    yellow;
    unsigned short    black;
};

/* CMY color value */
struct CMCMYColor {
    unsigned short    cyan;
    unsigned short    magenta;
    unsigned short    yellow;
};

/* HiFi color values */
struct CMMultichannel5Color {
    unsigned char    components[5];
};

```

```

struct CMMultichannel6Color {
    unsigned char    components[6];
};

struct CMMultichannel7Color {
    unsigned char    components[7];
};

struct CMMultichannel8Color {
    unsigned char    components[8];
};

/* gray color value */
struct CMGrayColor {
    unsigned short    gray;
};

/* color union */
union CMColor {
    CM                rgb;
    CMHSVColor        hsv;
    CMHLSColor        hls;
    CMXYZColor        XYZ;
    CMLabColor        Lab;
    CMLuvColor        Luv;
    CMYxyColor        Yxy;
    CMCMYKColor        cmyk;
    CMCMYColor        cmy;
    CMGrayColor        gray;
    CMMultichannel5Color mc5;
    CMMultichannel6Color mc6;
    CMMultichannel7Color mc7;
    CMMultichannel8Color mc8;
};

/* ColorSync Manager bitmap */
struct CMBitmap {
    char                *image;
    long                width;
    long                height;
    long                rowBytes;
    long                pixelSize;
};

```

Developing ColorSync-Supportive Applications

```

    CBitmapColorSpace  space;
    long               user1;
    long               user2;
};

/* profile reference abstract data type */
typedef struct CMPrivateProfileRecord *CMProfileRef;

/* profile search result reference abstract data type */
struct CMPrivateProfileSearchResult *CMProfileSearchRef;

/* high-level color-matching session reference abstract data type */
struct CMPrivateMatchRefRecord *CMMatchRef;

/* color world reference abstract data type */
struct CMPrivateColorWorldRecord *CMWorldRef;

/* TEnableColorMatchingBlk */
struct TEnableColorMatchingBlk {
    short          iOpCode;
    short          iError;
    long           lReserved;
    THPrint        hPrint;
    Boolean        fEnableIt;
    SInt8          filler;
};

/* ColorSync version 1.0 profile header */
struct CMHeader {
    unsigned long   size;
    OSType          CMType;
    unsigned long   applProfileVersion;
    OSType          dataType;
    OSType          deviceType;
    OSType          deviceManufacturer;
    unsigned long   deviceModel;
    unsigned long   deviceAttributes[2];
    unsigned long   profileNameOffset;
    unsigned long   customDataOffset;
    CMMatchFlag     flags;
    CMMatchOption   options;
};

```



```

    CMXYZColor      white;
    CMXYZColor      black;
};

/* PostScript color rendering dictionary (CRD) virtual memory size tag structure */
struct CMIntentCRDVMSize {
    long             rendering    Intent;
    unsigned long     VMSize;
};

struct CMPS2CRDVMSizeType {
    OSType            typeDescriptor;
    unsigned long     reserved;
    unsigned long     count;
    CMIntentCRDVMSize intentCRD[1];
};

```

Functions

Accessing Profile Files

```

pascal CMError CMOpenProfile      (CMPProfileRef *prof, const CMPProfileLocation
                                   *theProfile);

pascal CMError CMCloseProfile     (CMPProfileRef prof);

pascal CMError CMUpdateProfile    (CMPProfileRef prof);

pascal CMError CMNewProfile       (CMPProfileRef *prof, const CMPProfileLocation
                                   *theProfile);

pascal CMError CMCopyProfile      (CMPProfileRef *targetProf, const CMPProfileLocation
                                   *targetLocation, CMPProfileRef prof);

pascal CMError CMValidateProfile  (CMPProfileRef prof, Boolean *valid,
                                   Boolean *preferredCMMnotfound);

pascal CMError CMGetProfileLocation (
                                   CMPProfileRef prof,
                                   CMPProfileLocation *theProfile);

```

```

pascal CMError CMFlattenProfile (CMPProfileRef prof,
                                unsigned long flags,
                                CMFlattenUPP proc,
                                void *refCon,
                                Boolean *preferredCMMnotfound);

pascal CMError CMUnflattenProfile (FSSpec *resultFileSpec,
                                   CMFlattenUPP proc, void *refCon,
                                   Boolean *preferredCMMnotfound);

```

Accessing Profile Elements

```

pascal CMError CMPProfileElementExists (
                                CMPProfileRef prof,
                                OSType tag, Boolean *found);

pascal CMError CMCountProfileElements (
                                CMPProfileRef prof, unsigned long *elementCount);

pascal CMError CMGetProfileElement (
                                CMPProfileRef prof, OSType tag, unsigned long
                                *elementSize, void *elementData);

pascal CMError CMGetProfileHeader (CMPProfileRef prof, CMAAppleProfileHeader *header);

pascal CMError CMGetPartialProfileElement (
                                CMPProfileRef prof, OSType tag,
                                unsigned long offset, unsigned long *byteCount,
                                void *elementData);

pascal CMError CMSetProfileElementSize (
                                CMPProfileRef prof, OSType tag, unsigned long
                                elementSize);

pascal CMError CMGetIndProfileElementInfo (
                                CMPProfileRef prof, unsigned long index,
                                OSType *tag, unsigned long *elementSize,
                                Boolean *refs);

pascal CMError CMGetIndProfileElement (
                                CMPProfileRef prof, unsigned long index,
                                unsigned long *elementSize, void *elementData);

```

```

pascal CMError CMSetPartialProfileElement (
    CMPProfileRef prof, OSType tag,
    unsigned long offset, unsigned long byteCount,
    void *elementData);

pascal CMError CMSetProfileElement (
    CMPProfileRef prof, OSType tag,
    unsigned long elementSize, void *elementData);

pascal CMError CMSetProfileHeader (CMPProfileRef prof,
    const CMAPAppleProfileHeader *header);

pascal CMError CMSetProfileElementReference (
    CMPProfileRef prof,
    OSType elementTag, OSType referenceTag);

pascal CMError CMRemoveProfileElement (
    CMPProfileRef prof, OSType tag);

pascal CMError CMGetScriptProfileDescription (
    CMPProfileRef prof, Str255 name, ScriptCode *code);

```

Matching Colors Using QuickDraw Operations

```

pascal CMError NCMBeginMatching (CMPProfileRef src, CMPProfileRef dst, CMMatchRef
    *myRef);

pascal void CMEndMatching (CMMatchRef myRef);

pascal void CMEnableMatchingComment (
    Boolean enableIt);

```

Using Embedded Profiles With QuickDraw

```

pascal void NCMDrawMatchedPicture (
    PicHandle myPicture, CMPProfileRef dst,
    Rect *myRect);

pascal CMError NCMUseProfileComment (
    CMPProfileRef prof, unsigned long flags);

```

Matching Colors Using the Low-Level Functions

```

pascal CMError NCWNewColorWorld (CMWorldRef *cw, CMProfileRef src, CMProfileRef dst);
pascal CMError CWConcatColorWorld (CMWorldRef *cw, CMConcatProfileSet *profileSet);
pascal CMError CWNewLinkProfile (CMProfileRef *prof, const CMProfileLocation
                                *targetLocation, CMConcatProfileSet *profileSet);
pascal void CWDisposeColorWorld (CMWorldRef cw);
pascal CMError CWMatchPixMap (CMWorldRef cw, PixMap *myPixMap,
                              CMBitmapCallBackUPP progressProc, void *refCon);
pascal CMError CWCheckPixMap (CMWorldRef cw, PixMap *myPixMap,
                              CMBitmapCallBackUPP progressProc, void *refCon,
                              BitMap *resultBitMap);
pascal CMError CWMatchBitmap (CMWorldRef cw, CMBitmap *bitMap,
                              CMBitmapCallBackUPP progressProc, void *refCon,
                              CMBitmap *matchedBitMap);
pascal CMError CWCheckBitmap (CMWorldRef cw, const CMBitmap *bitMap,
                              CMBitmapCallBackUPP progressProc,
                              void *refCon, CMBitmap *resultBitMap);
pascal CMError CWMatchColors (CMWorldRef cw, CMColor *myColors, unsigned long
                              count);
pascal CMError CWCheckColors (CMWorldRef cw, CMColor *myColors,
                              unsigned long count, long *result);

```

Assigning and Accessing the System Profile File

```

pascal CMError CMSetSystemProfile (const FSSpec *profileFileSpec);
pascal CMError CMGetSystemProfile (CMProfileRef *prof);

```

Searching External Profiles

```

pascal CMError CMNewProfileSearch (CMSearchRecord *searchSpec, void *refCon,
                                   unsigned long *count, CMProfileSearchRef
                                   *searchResult);
pascal CMError CMUpdateProfileSearch (
                                   CMProfileSearchRef search,
                                   void *refCon, unsigned long *count);

```

```

pascal void CMDisposeProfileSearch (
    CMPProfileSearchRef search);

pascal CMError CMSearchGetIndProfile (
    CMPProfileSearchRef search,
    unsigned long index, CMPProfileRef *prof);

pascal CMError CMSearchGetIndProfileFileSpec (
    CMPProfileSearchRef search, unsigned long index,
    FSSpec *profileFile);

```

Converting Between Color Spaces

```

pascal ComponentResult CMXYZToLab (ComponentInstance ci, const CMColor *src,
    const CMXYZColor *white, CMColor *dst,
    unsigned long count);

pascal ComponentResult CMLabToXYZ (ComponentInstance ci, const CMColor *src,
    const CMXYZColor *white, CMColor *dst,
    unsigned long count);

pascal ComponentResult CMXYZToLuv (ComponentInstance ci, const CMColor *src,
    const CMXYZColor *white, CMColor *dst,
    unsigned long count);

pascal ComponentResult CMLuvToXYZ (ComponentInstance ci, const CMColor *src,
    const CMXYZColor *white, CMColor *dst,
    unsigned long count);

pascal ComponentResult CMXYZToYxy (ComponentInstance ci, const CMColor *src,
    CMColor *dst, unsigned long count);

pascal ComponentResult CMYxyToXYZ (ComponentInstance ci, const CMColor *src,
    CMColor *dst, unsigned long count);

pascal ComponentResult CMXYZToFixedXYZ (
    ComponentInstance ci, const CMXYZColor *src,
    CMFixedXYZColor *dst, unsigned long count);

pascal ComponentResult CMFixedXYZToXYZ (
    ComponentInstance ci,
    const CMFixedXYZColor *src,
    CMXYZColor *dst,
    unsigned long count);

```

Developing ColorSync-Supportive Applications

```

pascal ComponentResult CMRGBToHLS (ComponentInstance ci,
                                   const CMColor *src,
                                   CMColor *dst,
                                   unsigned long count);

pascal ComponentResult CMHLSToRGB (ComponentInstance ci,
                                   const CMColor *src,
                                   CMColor *dst,
                                   unsigned long count);

pascal ComponentResult CMRGBToHSV (ComponentInstance ci,
                                   const CMColor *src,
                                   CMColor *dst,
                                   unsigned long count);

pascal ComponentResult CMHSVToRGB (ComponentInstance ci,
                                   const CMColor *src,
                                   CMColor *dst, unsigned long count);

pascal ComponentResult CMRGBToGray (
    ComponentInstance ci,
    const CMColor *src,
    CMColor *dst,
    unsigned long count);

```

PostScript Color-Matching Support Functions

```

pascal CLError CMGetPS2ColorSpace (CMPProfileRef srcProf,
                                   unsigned long flags,
                                   CMFlattenUPP proc, void *refCon,
                                   Boolean *preferredCMMnotfound);

pascal CLError CMGetPS2ColorRenderingIntent (
    CMPProfileRef srcProf,
    unsigned long flags,
    CMFlattenUPP proc, void *refCon,
    Boolean *preferredCMMnotfound);

pascal CLError CMGetPS2ColorRendering (
    CMPProfileRef srcProf,
    CMPProfileRef dstProf,
    unsigned long flags,
    CMFlattenUPP proc, void *refCon,
    Boolean *preferredCMMnotfound);

```

```
extern pascal CMError CMGetPS2ColorRenderingVMSize (
    CMProfileRef srcProf, CMProfileRef dstProf,
    unsigned long *vmSize,
    Boolean *preferredCMMnotfound);
```

Locating the ColorSync™ Profiles Folder

```
pascal CMError CMGetColorSyncFolderSpec (
    short vRefNum,
    Boolean createFolder,
    short *foundVRefNum,
    long *foundDirID);
```

Obtaining Information About a Color World

```
pascal CMError CMGetCWInfo (CMWorldRef cw, CMCWInfoRecord *info);
```

Application-Supplied Functions for the ColorSync Manager

```
pascal OSErr MyColorSyncDataTransfer (
    long command, long *size,
    void *data, void *refCon);

pascal Boolean MyCMBitmapCallBackProc (
    long progress,
    void *refCon);

pascal Boolean MyCMProfileFilterProc (
    CMProfileRef prof,
    void *refCon);
```


Developing Color Management Modules

Contents

About Color Management Modules	5-4
Creating a Color Management Module	5-6
Creating a Component Resource for a CMM	5-6
How Your CMM Is Called by the Component Manager	5-9
Handling Request Codes	5-10
Responding to Required Component Manager Request Codes	5-21
Responding to ColorSync Manager Required Request Codes	5-22
Responding to ColorSync Manager Optional Request Codes	5-25
Summary of the Color Management Modules	5-39
Constants	5-39
Functions	5-40

This chapter describes how to create a component called a color management module (CMM) that can be used with the ColorSync Manager instead of or in conjunction with the Apple-supplied default CMM. At a minimum, a CMM created for use with the ColorSync Manager must be able to match colors across color spaces belonging to different base families and check colors expressed in the color gamut of one device against the color gamut of another device.

In addition to the minimum set of requests a CMM must service, a CMM can also implement support for other requests a ColorSync-supportive application or device driver might make. Among the optional services a CMM might provide are verifying if a particular profile contains the base set of required elements for a profile of its type and directing the process of converting profile data embedded in a graphics file to data in an external profile file accessed through a profile reference and vice versa. A CMM can also provide services for PostScript™ printers by obtaining or deriving from a profile specific data required by PostScript printers for color-matching processes and returning the data in a format that can be sent to the PostScript printer.

You should read this chapter if you are a third-party developer who creates CMMs for use with version 2.0 of the ColorSync Manager.

This chapter gives a brief overview of what a CMM is and the role a CMM plays in the ColorSync Manager color management system (CMS). Before reading this chapter, you should read the chapter “Introduction to the ColorSync Manager” in this book for a more complete conceptual explanation of how a CMM fits within the ColorSync Manager CMS.

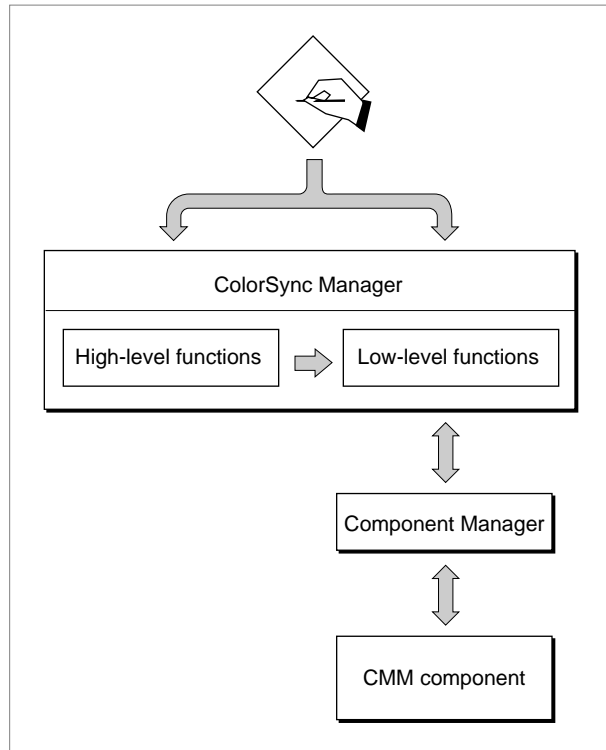
This chapter provides you with a basic structure you can follow in creating a CMM and a high-level discussion of the required and optional ColorSync Manager request codes your CMM might be called to handle, as well as the Component Manager required request codes to which every component must respond.

For complete details on components and their structure, see the chapter “Component Manager” in *Inside Macintosh: More Macintosh Toolbox*.

About Color Management Modules

A color management module (CMM) is a component that implements color matching, color gamut checking, and other services and performs these services in response to requests from ColorSync-supportive applications or device drivers.

A CMM component interacts directly with the Component Manager, which calls the CMM on behalf of the ColorSync Manager and the requesting application or driver. When they call ColorSync Manager functions to request color-matching and color-checking services, ColorSync-supportive applications and device drivers specify the profiles to be used. These profiles characterize the devices involved; they include information giving the color spaces and the color gamuts of the devices and the preferred CMM to be used to carry out the work. A CMM uses the information contained in these profiles to perform the processing required to service requests. Figure 5-1 shows the relationship between a ColorSync-supportive application or driver, the ColorSync Manager, the Component Manager, and a CMM.

Figure 5-1 The ColorSync Manager and the Component Manager

A CMM should support all six classes of profiles defined by the ICC. For information on the six classes of profiles, see the chapter “ColorSync Manager Reference for Applications and Device Drivers” on the enclosed CD and the *International Color Consortium Profile Format Specification*, version 2.0, document revision 3.x. For information on how to obtain a copy of this document, contact the Developer Support organization of Apple Computer. See the preface of this book for information explaining how to contact Developer Support.

In some cases, a CMM will not be able to convert and match colors directly from the color space of one profile to that of another. Instead, it will need to convert colors to the device-independent color space specified by the profile. Device-independent color spaces, or interchange color spaces, are used for the interchange of color data from the native color space of one device to the native

color space of another device. The profile connection space field of a profile header specifies the interchange color space for that profile. Version 2.0 of the ColorSync Manager supports two interchange color spaces: XYZ and Lab.

When interchange color spaces are involved, the ColorSync Manager handles the process, which is largely transparent to the CMM. The ColorSync Manager passes to the CMM the correct profiles for color matching. For example, in a case in which both the source and destination profile's CMMs are required to complete the color matching using color space profiles, the ColorSync Manager calls the source profile's CMM with the source profile and an interchange color space profile. Then it calls the destination profile's CMM with an interchange color space profile and the destination profile. The ColorSync Manager assesses the requirements and breaks the process down so that the correct CMM is called with the correct set of profiles. This process is described from the perspective of an application or device driver in the chapter "Developing ColorSync-Supportive Applications."

A CMM uses lookup tables and algorithms for color matching, previewing color reproduction capabilities of one device on another, and checking for colors that cannot be reproduced.

Creating a Color Management Module

This section describes how to create a CMM component, including how to respond to required Component Manager and ColorSync Manager requests and optional ColorSync Manager requests.

Creating a Component Resource for a CMM

A CMM is stored as a component resource. It contains a number of resources, including the standard component resource (a resource of type `'thng'`) required of any Component Manager component. In addition, a CMM must contain code to handle required request codes passed to it by the Component Manager. This includes support for Component Manager required request codes as well as ColorSync Manager required request codes.

To allow the ColorSync Manager to use your CMM when a profile specifies it as its preferred CMM, your CMM should be located in the Extensions folder,

where it will automatically be registered at startup. The file type for component files must be set to 'thng'.

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. As part of the component resource, you must provide a component description record that specifies the component type, subtype, manufacturer, and flags. Here is the component description data structure:

```
struct ComponentDescription {
    OSType          componentType;
    OSType          componentSubType;
    OSType          componentManufacturer;
    unsigned long   componentFlags;
    unsigned long   componentFlagsMask;
};
```

The component description data structure of type `ComponentDescription` contains the following fields for which you must set values:

- The `componentType` field contains a unique 4-byte code specifying the resource type and resource ID of the component's executable code. For your CMM, set this field to 'cmm'.
- The `componentSubType` field indicates the type of services your CMM provides. You should set this field to your CMM name. This value must match exactly the value specified in the profile header's `CMMType` field. You must register this value with the ICC.
- The `componentManufacturer` field indicates the creator of the CMM. You may set this field to any value you wish.
- The `componentFlags` field is a 32-bit field that provides additional information about your CMM component. The high-order 8 bits are reserved for definition by the Component Manager. 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. For more information, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox*.
- The `componentFlagsMask` field is reserved.

Note

Values you specify for all fields except the `componentType` field must include at least one uppercase character. Apple Computer reserves values containing all lowercase characters for its own use. ♦

Listing 5-1 shows a Rez listing of a component resource that describes a CMM.

Listing 5-1 CMM component Rez listing

```

#define UseExtendedThingResource1
#include "Types.r"

resource 'STR ' (128, purgeable) {
    "CMM Component"
};

resource 'STR ' (129, purgeable) {
    "Copyright © 1995 Apple Computer, Inc."
};

resource 'ICN#' (128, purgeable) {
    { /* array: 2 elements */
        /* [1] */
        $"FFE0 07FF 8040 0201 8080 0101 807F FE01"
        $"8000 0001 8000 0001 8003 F001 800F FC01"
        $"803F FF01 807F FF01 807E 9E01 80F8 4401"
        $"80F4 8A01 81F2 5201 81E0 8101 81F5 5501"
        $"81EA AB01 81E0 4101 81F2 9201 80F4 4A01"
        $"80F8 8401 807E 5E01 807F FF01 803F FF01"
        $"800F FC01 8003 F001 8000 0001 8000 0001"
        $"FFF0 0FFF 0020 0400 0040 0200 003F FC",
        /* [2] */
        $"FFE0 07FF FFC0 03FF FF80 01FF FFFF FFFF"
        $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
        $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
        $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
        $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
        $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
    }
};

```



```

        $"FFFF FFFF 003F FC00 007F FE00 003F FC"
    }
};

resource 'thng' (128, purgeable) {
    'cmm ',
    'fake',
    'fake',
    0X80000000,
    kAnyComponentFlagsMask,
    'cmm ',
    128,
    'STR ',
    128,
    'STR ',
    129,
    'ICN#',
    128,
    0x0,
    9,
    128,
    { /* array ComponentPlatformInfo: 2 elements */
        /* [1] */
        0X80000000, 'cmm ', 128, platform68k,
        /* [2] */
        0X80000000, 'cmm ', 129, platformPowerPC
    }
};

```

How Your CMM Is Called by the Component Manager

Because a CMM is a direct client of the Component Manager, it must conform to the Component Manager's interface requirements, including supporting and responding to required Component Manager calls.

The code for your CMM should be contained in a resource. The Component Manager expects that the entry point to this resource is a function having this format:

```
pascal ComponentResult main(ComponentParameters *params, Handle storage);
```

Whenever the Component Manager receives a request for your CMM, it calls your component's entry point and passes any parameters, along with information about the current connection, in a component parameters data structure of type `ComponentParameters`. This entry point must be the first function in your CMM's code segment. The Component Manager also passes a handle to the private storage (if any) associated with the current instance of your component. Here is the component parameters data structure:

```
struct ComponentParameters {  
    unsigned char    flags;  
    unsigned char    paramSize;  
    short            what;  
    long             params[1];  
};
```

The first field of the component parameters data structure is reserved. The following three fields carry information your CMM needs to perform its processing. The `what` field contains a value that identifies the type of request. The `paramSize` field specifies the size in bytes of the parameters passed from the ColorSync-supportive calling application to your CMM. The parameters themselves are passed in the `params` field.

Handling Request Codes

When your component receives a request, it should examine the `what` field of the component parameters data structure 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.

Your component's entry point function should interpret the request code and possibly dispatch the request to some other subroutine. To streamline your CMM code, you can implement separate subroutines to handle each of the request codes you support. The chapter "ColorSync Manager Reference for Color Management Modules," in the *Advanced Color Imaging Reference* provided on the enclosed CD, describes the prototype for each function your CMM must supply in order to handle the corresponding ColorSync Manager request code.

At a minimum, your CMM must handle the required Component Manager and required ColorSync Manager request codes.

Your CMM component must be able to handle the required Component Manager request codes, defined by these constants:

- `kComponentOpenSelect (-1)`
Requests that you open an instance of the component. For more information, see “Establishing the Environment for a New Component Instance” on page 5-21.
- `kComponentCloseSelect (-2)`
Requests that you close the component instance. For more information, see “Releasing Private Storage and Closing the Component Instance” on page 5-21.
- `kComponentCanDoSelect (-3)`
Requests that you tell whether your CMM handles a specific request. For more information, see “Reporting If Your CMM Supports a Request” on page 5-22.
- `kComponentVersionSelect (-4)`
Requests that you return your CMM’s version number. For more information, see “Providing Your CMM Version Number” on page 5-22.

Your CMM must also be able to handle the required ColorSync Manager request codes defined by these constants:

- `kCMMInit (0)`
Requests that you initialize the current component instance of your CMM for a ColorSync 1.0 session. This is a required request code only if your CMM supports ColorSync 1.0 profiles.
- `kCMMMatchColors (1)`
Requests that you color match the specified colors from one color space to another. For more information, see “Matching a List of Colors to the Destination Profile’s Color Space” on page 5-23.
- `kCMMCheckColors (2)`
Requests that you check the specified colors against the gamut of the destination device whose profile is specified. For more information, see “Checking a List of Colors” on page 5-24.
- `kNCMMInit (6)`
Requests that you initialize the current component instance of your CMM for a ColorSync Manager 2.0 session. For more information, see “Initializing the Current Component Instance for a Session Involving Two Profiles” on page 5-23.

The Component Manager may also call your CMM with the following ColorSync Manager request codes that are considered optional. A CMM may support these requests, although you are not required to do so.

- `kCMMMatchPixMap` (3)
Requests that you match the colors of a pixel map image to the color gamut of a destination profile, replacing the original pixel colors with their corresponding colors. For more information, see “Matching the Colors of a Pixel Map Image” on page 5-30.
- `kCMMCheckPixMap` (4)
Requests that you check the colors of a pixel map image against the gamut of a destination device for inclusion and report the results. For more information, see “Checking the Colors of a Pixel Map Image” on page 5-31.
- `kCMMConcatenateProfiles` (5)
This request code is for backward compatibility with ColorSync 1.0.
- `kCMMConcatInit` (7)
Requests that you initialize any private data your CMM will need for a color session involving the set of profiles specified by the profile array pointed to by the `profileSet` parameter. For more information, see “Initializing the Component Instance for a Session Using Concatenated Profiles” on page 5-31.
- `kCMMValidateProfile` (8)
Requests that you test a specific profile to determine if the profile contains the minimum set of elements required for a profile of its type. For more information, see “Validating That a Profile Meets the Base Content Requirements” on page 5-26.
- `kCMMMatchBitmap` (9)
Requests that you match the colors of a source image bitmap to the color gamut of a destination profile. For more information, see “Matching the Colors of a Bitmap” on page 5-27.
- `kCMMCheckBitmap` (10)
Requests that you check the colors of a source image bitmap against the color gamut of a destination profile. For more information, see “Checking the Colors of a Bitmap” on page 5-28.
- `kCMMGetPS2ColorSpace` (11)
Requests that you obtain or derive the color space data from a source profile and pass the data to a low-level data-transfer function supplied by the

calling application or device driver. For more information, see “Obtaining PostScript-Related Data From a Profile” on page 5-33.

- `kCMMGetPS2ColorRenderingIntent` (12)
Requests that you obtain the color-rendering intent from the header of a source profile and then pass the data to a low-level data-transfer function supplied by the calling application or device driver. For more information, see “Obtaining PostScript-Related Data From a Profile” on page 5-33.
- `kCMMGetPS2ColorRendering` (13)
Requests that you obtain the rendering intent from the source profile’s header, generate the color rendering dictionary (CRD) data from the destination profile, and then pass the data to a low-level data-transfer function supplied by the calling application or device driver. For more information, see “Obtaining PostScript-Related Data From a Profile” on page 5-33.
- `kCMMFlattenProfile` (14)
Requests that you extract profile data from the profile to be flattened and pass the profile data to a function supplied by the calling program. For more information, see “Flattening a Profile for Embedding in a Graphics File” on page 5-36.
- `kCMMUnflattenProfile` (15)
Requests that you create a file in the temporary items folder in which to store profile data you receive from a function. The calling program supplies the function. You call this function to obtain the profile data. For more information, see “Unflattening a Profile” on page 5-37.
- `kCMMNewLinkProfile` (16)
Requests that you create a single device-linked profile that includes the profiles passed to you in an array. For more information, see “Creating a Device-Linked Profile and Opening a Reference to It” on page 5-32.
- `kCMMGetPS2ColorRenderingVMSize` (17)
Requests that you obtain or assess the maximum virtual memory (VM) size of the color rendering dictionary (CRD) specified by a destination profile. For more information, see “Obtaining the Size of the Color Rendering Dictionary for PostScript Printers” on page 5-35.

After examining the `what` field to identify the request, your CMM code must extract the calling program’s parameters from the `params` field of the component parameters data structure and call your appropriate subroutine handler, passing the parameters to it. For a description of component

parameters data structure of type `ComponentParameters`, see “How Your CMM Is Called by the Component Manager,” beginning on page 5-9.

Your code can unpack these parameters itself, or it can call the Component Manager’s `CallComponentFunctionWithStorage` function or `CallComponentFunction` function to perform these services.

The `CallComponentFunctionWithStorage` function is useful if your CMM uses private storage. When you call this function, you pass it a handle to the storage for this component instance, the component parameters data structure, and the address of your subroutine handler. Each time it calls your entry point function, the Component Manager passes to your function the storage handle along with the component parameters data structure. For a description of how you associate private storage with a component instance, see “Establishing the Environment for a New Component Instance” on page 5-21. The Component Manager’s `CallComponentFunctionWithStorage` extracts the calling application’s parameters from the component parameters data structure and invokes your function, passing to it the extracted parameters and the private storage handle.

Listing 5-2 shows sample code that illustrates how to respond to the required Component Manager and ColorSync Manager requests. For a complete listing of the sample code on which this listing is based, see the technical note QT05 Component Manager 3.0. This technical note shows how to create a fat component, which is a single component usable for both 68K-based and PowerPC-based systems. The portion of sample code in Listing 5-2 for PowerPC-based systems uses a parameter block data structure to pass parameters. The data structure for this parameter block is generated from a macro in the `MixedMode.h` header file.

Listing 5-2 A CMM component shell

```
#include <Types.h>
#include <Quickdraw.h>
#include <Memory.h>
#include <Gestalt.h>
#include <Components.h>

#include <CMMComponent.h>
```

Developing Color Management Modules

```

#ifndef __powerc
#include <A4Stuff.h>
#endif

#define DEBUG 0

#ifndef DEBUG
#define DEBUG 0
#endif

/* component version */
#define CMCodeVersion 0
#define CMVersion ((CMMInterfaceVersion << 16) | CMCodeVersion)

/* component storage */
struct CMMStorageRecord {ComponentInstance ci;};
typedef struct CMMStorageRecord CMMStorageRecord, **CMMStorageHandle;

#if __powerc
#define CallComponentFunctionWithStorageUniv(storage, params, funcName) \
    CallComponentFunctionWithStorage(storage, params, &funcName##RD)
#define CallComponentFunctionUniv(params, funcName) \
    CallComponentFunction(params, &funcName##RD)
#define INSTANTIATE_ROUTINE_DESCRIPTOR(funcName) RoutineDescriptor funcName##RD = \
    BUILD_ROUTINE_DESCRIPTOR (upp##funcName##ProcInfo, funcName)
#else
#define CallComponentFunctionWithStorageUniv(storage, params, funcName) \
    CallComponentFunctionWithStorage(storage, params, \
        (ComponentFunctionUPP)funcName)
#define CallComponentFunctionUniv(params, funcName) \
    CallComponentFunction(params, (ComponentFunctionUPP)funcName)
#endif

enum{
    uppDoComponentOpenProcInfo = kPascalStackBased
        | RESULT_SIZE(SIZE_CODE(sizeof(ComponentResult)))
        | STACK_ROUTINE_PARAMETER(1,SIZE_CODE(sizeof(ComponentInstance)))
};

```

```

enum {
    uppDoComponentCloseProcInfo = kPascalStackBased
        | RESULT_SIZE(SIZE_CODE(sizeof(ComponentResult)))
        | STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(Handle)))
        | STACK_ROUTINE_PARAMETER(2, SIZE_CODE(sizeof(ComponentInstance)))
};

enum {
    uppDoComponentCanDoProcInfo = kPascalStackBased
        | RESULT_SIZE(SIZE_CODE(sizeof(ComponentResult)))
        | STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(short)))
};

enum {
    uppDoComponentVersionProcInfo = kPascalStackBased
        | RESULT_SIZE(SIZE_CODE(sizeof(ComponentResult)))
};

enum {
    uppDoComponentRegisterProcInfo = kPascalStackBased
        | RESULT_SIZE(SIZE_CODE(sizeof(ComponentResult)))
};

enum {
    uppDoCMInitProcInfo = kPascalStackBased
        | RESULT_SIZE(SIZE_CODE(sizeof(ComponentResult)))
        | STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(Handle)))
        | STACK_ROUTINE_PARAMETER(2, SIZE_CODE(sizeof(CMProfileHandle)))
        | STACK_ROUTINE_PARAMETER(3, SIZE_CODE(sizeof(CMProfileHandle)))
};

enum {
    uppDoNCMInitProcInfo = kPascalStackBased
        | RESULT_SIZE(SIZE_CODE(sizeof(ComponentResult)))
        | STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(Handle)))
        | STACK_ROUTINE_PARAMETER(2, SIZE_CODE(sizeof(CMProfileRef)))
        | STACK_ROUTINE_PARAMETER(3, SIZE_CODE(sizeof(CMProfileRef)))
};

```



```

enum {
    uppDoCMMatchColorsProcInfo = kPascalStackBased
        | RESULT_SIZE(SIZE_CODE(sizeof(ComponentResult)))
        | STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(Handle)))
        | STACK_ROUTINE_PARAMETER(2, SIZE_CODE(sizeof(CMColor*)))
        | STACK_ROUTINE_PARAMETER(3, SIZE_CODE(sizeof(unsigned long)))
};

enum {
    uppDoCMCheckColorsProcInfo = kPascalStackBased
        | RESULT_SIZE(SIZE_CODE(sizeof(ComponentResult)))
        | STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(Handle)))
        | STACK_ROUTINE_PARAMETER(2, SIZE_CODE(sizeof(CMColor*)))
        | STACK_ROUTINE_PARAMETER(3, SIZE_CODE(sizeof(unsigned long)))
        | STACK_ROUTINE_PARAMETER(4, SIZE_CODE(sizeof(long*)))
};

/* function prototypes */
pascal ComponentResult main(ComponentParameters *params, Handle storage);

pascal ComponentResult DoComponentOpen(ComponentInstance self);

pascal ComponentResult DoComponentClose(CMMStorageHandle storage,
                                         ComponentInstance self);

pascal ComponentResult DoComponentCanDo(short selector);

pascal ComponentResult DoComponentVersion(void);

pascal ComponentResult DoComponentRegister(void);

pascalComponentResult DoCMInit(CMMStorageHandle storage, CMProfileHandle srcProfile,
                               CMProfileHandle dstProfile);

pascalComponentResult DoNCMInit(CMMStorageHandle storage, CMProfileRef srcProfile,
                                CMProfileRef dstProfile);

pascalComponentResult DoCMMatchColors(CMMStorageHandle storage, CMColor *colorBuf,
                                       unsigned long count);

```

Developing Color Management Modules

```

pascal ComponentResult DoCMCheckColors(CMMStorageHandle storage, CMColor *colorBuf,
                                         unsigned long count, long *gamutResult);

#if __powerc
    /* Routine descriptors for component functions */
    INSTANTIATE_ROUTINE_DESCRIPTOR(DoComponentOpen);
    INSTANTIATE_ROUTINE_DESCRIPTOR(DoComponentClose);
    INSTANTIATE_ROUTINE_DESCRIPTOR(DoComponentCanDo);
    INSTANTIATE_ROUTINE_DESCRIPTOR(DoComponentVersion);
    INSTANTIATE_ROUTINE_DESCRIPTOR(DoComponentRegister);
    INSTANTIATE_ROUTINE_DESCRIPTOR(DoCMInit);
    INSTANTIATE_ROUTINE_DESCRIPTOR(DoNCMInit);
    INSTANTIATE_ROUTINE_DESCRIPTOR(DoCMMatchColors);
    INSTANTIATE_ROUTINE_DESCRIPTOR(DoCMCheckColors);

    /* PowerPC main component entry point */
    RoutineDescriptor MainRD = BUILD_ROUTINE_DESCRIPTOR(uppComponentRoutineProcInfo,
                                                         main);

    ProcInfoType __procinfo = uppComponentRoutineProcInfo;
#endif

    pascal ComponentResult main(ComponentParameters *params,
                                Handle storage);

Abstract:
    main entry point to CMM Component

Params:
    params (in)Parameters in form used by Component Manager
    storage (in)Handle to memory to be used by CMM

Return:
    noErr    If successful
            Otherwise System or ColorSync result code

/* This main function must run on all Macs. */
pascalComponentResult main(ComponentParameters *params, Handle storage)
{
    ComponentResult result;
    short           message;

```

```

#if !__powerc
long      oldA4;

oldA4 = SetCurrentA4();
#endif

message = (*params).what;

/* selectors < 0 for Component Manager functions */
if (message < 0)
{
    switch (message)
    {
        case kComponentOpenSelect :
            result = CallComponentFunctionUniv (params, DoComponentOpen);
            break;

        case kComponentCloseSelect :
            result = CallComponentFunctionWithStorageUniv(storage, params,
                                                         DoComponentClose);
            break;

        case kComponentCanDoSelect :
            result = CallComponentFunctionUniv(params, DoComponentCanDo);
            break;

        case kComponentVersionSelect :
            result = CallComponentFunctionUniv(params, DoComponentVersion);
            break;

        case kComponentRegisterSelect :
            result = CallComponentFunctionUniv(params, DoComponentRegister);
            break;

        default :
            result = noErr;
            break;
    }
}

```

Developing Color Management Modules

```

/* selectors >= 0 for CMM functions */
else
{
    switch (message)
    {
        case kCMMInit :
            result = CallComponentFunctionWithStorageUniv(storage, params,
                                                         DoCMMInit);

            break;
        case kNCMMInit :
            result = CallComponentFunctionWithStorageUniv(storage, params,
                                                         DoNCMInit);

            break;
        case kCMMMatchColors :
            result = CallComponentFunctionWithStorageUniv(storage, params,
                                                         DoCMMMatchColors);

            break;
        case kCMMCheckColors :
            result = CallComponentFunctionWithStorageUniv(storage, params,
                                                         DoCMMCheckColors);

            break;
        default :
            result = unimpErr;
            break;
    }
}

#ifdef __powerc
    SetA4(oldA4);
#endif

return (result);
}

```

For more information describing how your CMM component should respond to request code calls from the Component Manager, see the section “Creating Components” in the chapter “Component Manager” in *Inside Macintosh: More Macintosh Toolbox*.

Responding to Required Component Manager Request Codes

This section describes some of the processes your CMM can perform in response to the following Component Manager requests that it must handle:

- “Establishing the Environment for a New Component Instance” describes how to handle a `kComponentOpenSelect` request.
- “Releasing Private Storage and Closing the Component Instance” describes how to handle a `kComponentCloseSelect` request.
- “Reporting If Your CMM Supports a Request” describes how to handle a `kComponentCanDoSelect` request.
- “Providing Your CMM Version Number” describes how to handle a `kComponentVersionSelect` request.

Establishing the Environment for a New Component Instance

When a ColorSync-supportive application or device driver first calls a function that requires the services of your CMM, the Component Manager calls your CMM with a `kComponentOpenSelect` request to open and establish an instance of your component for the calling program. The component instance defines a unique connection between the calling program and your CMM.

In response to this request, you should allocate memory for any private data you require for the connection. You should allocate memory from the current heap zone. If that attempt fails, you should allocate memory from the system heap or the temporary heap. You can use the `SetComponentInstanceStorage` function to associate the allocated memory with the component instance.

For more information on how to respond to this request and open connections to other components, see the section “Creating Components” in the chapter “Component Manager” in *Inside Macintosh: More Macintosh Toolbox*.

Releasing Private Storage and Closing the Component Instance

To call your CMM with a close request, the Component Manager sets the `what` field of the component parameters data structure to `kComponentCloseSelect`. In response to this request code, your CMM should dispose of the storage memory associated with the connection.

Reporting If Your CMM Supports a Request

Before the ColorSync Manager calls your CMM with a request code on behalf of a ColorSync-supportive application or driver that called the corresponding function, the Component Manager calls your CMM with a can do request to determine if your CMM implements support for the request.

To call your CMM with a can do request, the Component Manager sets the `what` field of the component parameters data structure to the value `kComponentCanDoSelect`. In response, you should set your CMM entry point function's result to 1 if your CMM supports the request and 0 if it doesn't.

Providing Your CMM Version Number

To call your CMM requesting its version number, the Component Manager sets the `what` field of the component parameters data structure to the value `kComponentVersionSelect`. In response, you should set your CMM entry point function's result to the CMM version number. 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.

If your CMM supports the ColorSync Manager version 2.0, your CMM should return the constant for the major version defined by the following enumeration when the Component Manager calls your CMM with the `kComponentVersionSelect` request code:

```
enum {
    CMMInterfaceVersion = 1
};
```

Responding to ColorSync Manager Required Request Codes

This section describes some of the processes your CMM can perform in response to the following ColorSync Manager requests that it must handle:

- “Initializing the Current Component Instance for a Session Involving Two Profiles” describes how to handle the `kNCMMInit` request.
- “Matching a List of Colors to the Destination Profile's Color Space” describes how to handle a `kCMMMatchColors` request.
- “Checking a List of Colors” describes how to handle a `kCMMCheckColors` request.

Initializing the Current Component Instance for a Session Involving Two Profiles

The Component Manager calls your CMM with an initialization request, setting the `what` field of the component parameters data structure to `kNCMMInit`. In most cases the Component Manager calls your CMM with an initialization request before it calls your CMM with any other ColorSync Manager requests.

In response to this request, your CMM should call its `MyNCMInit` initialization subroutine. For a description of the function prototype your initialization subroutine must adhere to, see the chapter “ColorSync Manager Reference for Color Management Modules” chapter of the *Advanced Color Imaging Reference* provided on the enclosed CD.

Using the private storage you allocated in response to the open request, your initialization subroutine should instantiate any private data it needs for the component instance. Before your entry point function returns a function result to the Component Manager, your subroutine should store any profile information it requires. In addition to the standard profile information, you should store the profile header’s quality flags setting, the profile size, and the rendering intent. After you return control to the Component Manager, you cannot use the profile references again.

This request gives you the opportunity to examine the profile contents before storing them. If you do not support some aspect of the profile, then you should return an unimplemented error in response to this request. For example, if your CMM does not implement multichannel color support, you should return an “unimplemented” error at this point.

The Component Manager may call your CMM with the `kNCMMInit` request code multiple times after it calls your CMM with a request to open the CMM. For example, it may call your CMM with an initialization request once with one pair of profiles and then again with another pair of profiles. For each call, you need to reinitialize the storage based on the content of the current profiles.

Your CMM should support all six classes of profiles defined by the ICC. For information on the six classes of profiles, see the chapter “ColorSync Manager Reference for Applications and Device Drivers” on the enclosed CD.

Matching a List of Colors to the Destination Profile’s Color Space

When a ColorSync-supportive application or device driver calls the `CWMatchColors` function for your CMM to handle, the Component Manager calls your CMM with a color-matching session request, setting the `what` field of the component parameters data structure to `kCMMMatchColors` and passing you a list

of colors to be matched. The Component Manager may also call your CMM with this request code to handle other cases, for example, when a ColorSync-supportive program calls the `CWMatchPixaMap` function.

Before it calls your CMM with this request, the Component Manager calls your CMM with one of the initialization requests—`kCMMInit`, `kNCMMInit`, or `kCMMConcatInit`—passing to your CMM in the `params` field of the component parameters data structure the profiles to be used for the color-matching session.

In response to the `kCMMMatchColors` request, your CMM should call its `MyCMMMatchColors` subroutine by calling the Component Manager's `CallComponentFunctionWithStorage` function and passing it a handle to the storage for this component instance, the component parameters data structure, and the address of your `MyCMMMatchColors` subroutine. For a description of the function prototype to which your subroutine must adhere, see the chapter “ColorSync Manager Reference for Color Management Modules” in the *Advanced Color Imaging Reference* provided on the enclosed CD.

The parameters passed to your CMM for this request include an array of type `CMColor` containing the list of colors to be matched and a one-based count of the number of colors in the list.

To handle this request, your CMM must match the source colors in the list to the color gamut of the destination profile, replacing the color value specifications in the `myColors` array with the matched colors specified in the destination profile's data color space. You should use the rendering intent and the quality flag setting of the source profile in matching the colors. For a description of the color list array data structure, see the section “The Color Union” in the “ColorSync Manager Reference for Applications and Device Drivers” chapter of the *Advanced Color Imaging Reference* provided on the enclosed CD.

Checking a List of Colors

When a ColorSync-supportive application or device driver calls the `CWCheckColors` function for your CMM to handle, the Component Manager calls your CMM with a color gamut-checking session request, setting the `what` field of the component parameters data structure to `kCMMCheckColors` and passing you a list of colors to be checked.

Before it calls your CMM with this request, the Component Manager calls your CMM with one of the initialization requests—`kCMMInit`, `kNCMMInit`, or `kCMMConcatInit`—passing to your CMM in the `params` field of the component

parameters data structure the profiles to be used for the color gamut-checking session.

In response to the `kCMMCheckColors` request, your CMM should call its `MyCMCheckColors` subroutine. For example, if you use the Component Manager's `CallComponentFunctionWithStorage` function, you would call it, passing it a handle to the storage for this component instance, the component parameters data structure, and the address of your `MyCMCheckColors` subroutine. For a description of the function prototype to which your subroutine must adhere, see the "ColorSync Manager Reference for Color Management Modules" chapter of the *Advanced Color Imaging Reference* provided on the enclosed CD.

In addition to the handle to the private storage containing the profile data, the `CallComponentFunctionWithStorage` function passes to your `MyCMCheckColors` subroutine an array of type `CMColor` containing the list of colors to be gamut checked, a one-based count of the number of colors in the list, and an array of longs.

To handle this request, your CMM should test the given list of colors against the gamut specified by the destination profile to report if the colors fall within a destination device's color gamut. For each source color in the list that is out of gamut, you must set the corresponding bit in the result array to 1.

Responding to ColorSync Manager Optional Request Codes

This section describes some of the processes your CMM can perform in response to the optional ColorSync Manager requests if your CMM supports them. Before the Component Manager calls your CMM with any of these requests, it first calls your CMM with a can do request to determine if you support the specific optional request code. This section includes the following:

- "Validating That a Profile Meets the Base Content Requirements" on page 5-26 describes how to handle a `kCMMValidateProfile` request.
- "Matching the Colors of a Bitmap" on page 5-27 describes how to handle a `kCMMMatchBitmap` request.
- "Checking the Colors of a Bitmap" on page 5-28 describes how to handle a `kCMMCheckBitmap` request.
- "Matching the Colors of a Pixel Map Image" on page 5-30 describes how to handle the `kCMMMatchPixMap` request.

Developing Color Management Modules

- “Checking the Colors of a Pixel Map Image” on page 5-31 describes how to handle the `kCMMCheckPixMap` request.
- “Initializing the Component Instance for a Session Using Concatenated Profiles” on page 5-31 describes how to handle a `kCMMConcatInit` request.
- “Creating a Device-Linked Profile and Opening a Reference to It” on page 5-32 describes how to handle a `kCMMNewLinkProfile` request.
- “Obtaining PostScript-Related Data From a Profile,” beginning on page 5-33 describes how to handle the `kCMMGetPS2ColorSpace`, `kCMMGetPS2ColorRenderingIntent`, and `kCMMGetPS2ColorRendering` requests.
- “Obtaining the Size of the Color Rendering Dictionary for PostScript Printers” on page 5-35 describes how to handle a `kCMMGetPS2ColorRenderingVMSize` request.
- “Flattening a Profile for Embedding in a Graphics File” on page 5-36 describes how to handle a `kCMMFlattenProfile` request.
- “Unflattening a Profile” on page 5-37 describes how to handle a `kCMMUnflattenProfile` request.

Validating That a Profile Meets the Base Content Requirements

When a ColorSync-supportive application or device-driver calls the `CMValidateProfile` function for your CMM to handle, the Component Manager calls your CMM with the `what` field of the component parameters data structure set to `kCMMValidateProfile` if your CMM supports the request.

In response to this request code, your CMM should call its `MyCMMValidateProfile` subroutine. One way to do this, for example, is by calling the Component Manager’s `CallComponentFunction` function, passing it the component parameters data structure and the address of your `MyCMMValidateProfile` subroutine. To handle this request, you don’t need private storage for ColorSync profile information, because the profile reference is passed to your function. However, if your CMM uses private storage for other purposes, you should call the Component Manager’s `CallComponentFunctionWithStorage` function. For a description of the function prototype to which your subroutine must adhere, see the “ColorSync Manager Reference for Color Management Modules” chapter of the *Advanced Color Imaging Reference* provided on the enclosed CD.

The `CallComponentFunction` function passes to your `MyCMMValidateProfile` subroutine a reference to the profile whose contents you must check and a flag whose value you must set to report the results.

To handle this request, your CMM should test the profile contents against the baseline profile elements requirements for a profile of this type as specified by the International Color Consortium. It should determine if the profile contains the minimum set of elements required for its type and set the response flag to `true` if the profile contains the required elements and `false` if it doesn't.

For information on how to obtain a copy of the *International Color Consortium Profile Format Specification*, version 2.0, document revision 3.x, contact the Developer Support organization of Apple Computer. See the preface of this book for information on how to contact Developer Support.

The ICC also defines optional tags, which may be included in a profile. Your CMM might use these optional elements to optimize or improve its processing. Additionally, a profile might include private tags defined to provide your CMM with processing capability it uses. The profile developer can define these private tags, register the tag signatures with the ICC, and include the tags in a profile.

If your CMM is dependent on optional or private tags, your `MyCMMValidateProfile` function should check for the existence of these tags also.

Instead of itself checking the profile for the minimum profile elements requirements for the profile type, your `MyCMMValidateProfile` function may use the Component Manager functions to call the default Apple-supplied CMM and have it perform the minimum defaults requirements validation.

To call the Apple-supplied CMM when responding to a `kCMMValidateProfile` request from an application, your CMM can use the standard mechanisms used by applications to call another component. For information on these mechanisms, see the chapter “Component Manager” in *Inside Macintosh: More Macintosh Toolbox*.

Matching the Colors of a Bitmap

When a ColorSync-supportive application or device driver calls the `CWMatchBitmap` function for your CMM to handle, the Component Manager calls your CMM with the `what` field of the component parameters data structure set to `kCMMMatchBitmap` if your CMM supports the request. If your CMM supports this request code, your CMM should be prepared to receive any of the bitmap types defined by the ColorSync Manager.

In response to this request code, your CMM should call its `MyCMMatchBitmap` subroutine. For example, to do this, your CMM may call the Component Manager's `CallComponentFunctionWithStorage` function, passing it the storage handle for this component instance, the component parameters data structure, and the address of your `MyCMMatchBitmap` subroutine. For a description of the function prototype to which your subroutine must adhere, see the “ColorSync Manager Reference for Color Management Modules” chapter of the *Advanced Color Imaging Reference* provided on the enclosed CD.

In addition to the storage handle for private storage for this component instance, the `CallComponentFunctionWithStorage` function passes to your `MyCMMatchBitmap` subroutine a pointer to the bitmap containing the source image data whose colors your function must match, a pointer to a callback function supplied by the calling program, a reference constant your subroutine must pass to the callback function when you invoke it, and a pointer to a bitmap in which your function stores the resulting color-matched image.

The callback function supplied by the calling function monitors the color-matching progress as your function matches the bitmap colors. You should call this function at regular intervals. Your `MyCMMatchBitmap` function should monitor the progress function for a returned value of `true`, which indicates that the user interrupted the color-matching process. In this case, you should terminate the color-matching process.

To handle this request, your `MyCMMatchBitmap` function must match the colors of the source image bitmap to the color gamut of the destination profile using the profiles specified by a previous `kNCMInit`, `kCMMInit`, or `kCMMConcatInit` request to your CMM for this component instance. You must store the color-matched image in the bitmap result parameter passed to your subroutine. If you are passed a `NULL` parameter, you must match the bitmap in place.

For a description of the prototype of the callback function supplied by the calling program, see the `MyCMBitmapCallbackProc` entry in the “ColorSync Manager Reference for Applications and Device Drivers” chapter of the *Advanced Color Imaging Reference* provided on the enclosed CD.

Checking the Colors of a Bitmap

When a ColorSync-supportive application or device driver calls the `CWCheckBitmap` function for your CMM to handle, the Component Manager calls your CMM with the `what` field of the component parameters data structure set to `kCMMCheckBitmap` if your CMM supports the request. If your CMM supports

this request code, your CMM should be prepared to receive any of the bitmap types defined by the ColorSync Manager.

In response to this request code, your CMM should call its `MyCMCheckBitmap` subroutine. For example, to do this, your CMM may call the Component Manager's `CallComponentFunctionWithStorage` function, passing it the storage handle for this component instance, the component parameters data structure, and the address of your `MyCMCheckBitmap` subroutine. For a description of the function prototype to which your subroutine must adhere, see the “ColorSync Manager Reference for Color Management Modules” chapter of the *Advanced Color Imaging Reference* provided on the enclosed CD.

In addition to the storage handle for private storage for this component instance, the `CallComponentFunctionWithStorage` function passes to your `MyCMCheckBitmap` subroutine a pointer to the bitmap containing the source image data whose colors your function must check, a pointer to a callback progress-reporting function supplied by the calling program, a reference constant your subroutine must pass to the callback function when you invoke it, and a pointer to a resulting bitmap whose pixels your subroutine must set to show if the corresponding source color is in or out of gamut.

The callback function supplied by the calling function monitors the color gamut-checking progress. You should call this function at regular intervals. Your `MyCMCheckBitmap` function should monitor the progress function for a returned value of `true`, which indicates that the user interrupted the color-checking process. In this case, you should terminate the process.

For a description of the prototype of the callback function supplied by the calling program, see the `MyCMBitmapCallBackProc` entry in the “ColorSync Manager Reference for Applications and Device Drivers” chapter in the *Advanced Color Imaging Reference* provided on the enclosed CD.

Using the content of the profiles that you stored at initialization time for this component instance, your `MyCMCheckBitmap` subroutine must check the colors of the source image bitmap against the color gamut of the destination profile. If a pixel is out of gamut, your function must set the corresponding pixel in the result image bitmap to 1. The ColorSync Manager returns the resulting bitmap to the calling application or driver to report the outcome of the check.

For complete details on the `MyCMCheckBitmap` subroutine parameters and how your `MyCMCheckBitmap` subroutine communicates with the callback function, see the `MyCMCheckBitmap` entry in the “ColorSync Manager Reference for Color Management Modules” chapter of the *Advanced Color Imaging Reference* provided on the enclosed CD.

Matching the Colors of a Pixel Map Image

When a ColorSync-supportive application or device driver calls the `CWMatchPixelFormat` function for your CMM to handle, the Component Manager calls your CMM with the `what` field of the component parameters data structure set to `kCMMMatchPixelFormat` if your CMM supports the request. If your CMM supports this request code, your `MyCMMMatchPixelFormat` function should be prepared to receive any of the pixel map types defined by QuickDraw.

In response to this request code, your CMM should call its `MyCMMMatchPixelFormat` subroutine. For example, to do this, your CMM may call the Component Manager's `CallComponentFunctionWithStorage` function, passing it the storage handle for this component instance, the component parameters data structure, and the address of your `MyCMMMatchPixelFormat` subroutine. For a description of the function prototype to which your subroutine must adhere, see the "ColorSync Manager Reference for Color Management Modules" chapter of the *Advanced Color Imaging Reference* provided on the enclosed CD.

In addition to the storage handle for private storage for this component instance, the `CallComponentFunctionWithStorage` function passes to your `MyCMMMatchPixelFormat` subroutine a pointer to the pixel map containing the source image to be matched, a pointer to a callback progress-reporting function supplied by the calling program, and a reference constant your subroutine must pass to the callback function when you invoke it.

To handle this request, your `MyCMMMatchPixelFormat` subroutine must match the colors of the source pixel map image to the color gamut of the destination profile, replacing the original pixel colors of the source image with their corresponding colors expressed in the data color space of the destination profile. The ColorSync Manager returns the resulting color-matched pixel map to the calling application or driver.

The callback function supplied by the calling function monitors the color-matching progress. You should call this function at regular intervals. Your `MyCMMMatchPixelFormat` function should monitor the progress function for a returned value of `true`, which indicates that the user interrupted the color-matching process. In this case, you should terminate the process.

For a description of the prototype of the callback function supplied by the calling program see the description of the function `MyCMBitmapCallbackProc` in the "ColorSync Manager Reference for Applications and Device Drivers" chapter of the *Advanced Color Imaging Reference*.

Checking the Colors of a Pixel Map Image

When a ColorSync-supportive application or device-driver calls the `CWCheckPixMap` function for your CMM to handle, the Component Manager calls your CMM with the `what` field of the component parameters data structure set to `kCMMCheckPixMap` if your CMM supports the request.

In response to this request code, your CMM should call its `MyCMCheckPixMap` subroutine. For example, to do this, your CMM may call the Component Manager's `CallComponentFunctionWithStorage` function, passing it the storage handle for this component instance, the component parameters data structure, and the address of your `MyCMCheckPixMap` subroutine. For a description of the function prototype to which your subroutine must adhere, see the "ColorSync Manager Reference for Color Management Modules" chapter of the *Advanced Color Imaging Reference* provided on the enclosed CD.

In addition to the storage handle for private storage for this component instance, the `CallComponentFunctionWithStorage` function passes to your `MyCMCheckPixMap` subroutine a pointer to the pixel map containing the source image to be checked, a QuickDraw bitmap in which to report the color-checking results, a pointer to a callback progress-reporting function supplied by the calling program, and a reference constant your subroutine must pass to the callback function when you invoke it.

Using the content of the profiles passed to you at initialization time, your `MyCMCheckPixMap` subroutine must check the colors of the source pixel map image against the color gamut of the destination profile to determine if the pixel colors are within the gamut. If a pixel is out of gamut, your subroutine must set to 1 the corresponding pixel of the result bitmap. The ColorSync Manager returns the bitmap showing the color-checking results to the calling application or device driver.

Initializing the Component Instance for a Session Using Concatenated Profiles

When a ColorSync-supportive application or device driver calls the `CWConcatColorWorld` function for your CMM to handle, the Component Manager calls your CMM with the `what` field of the component parameters data structure set to `kCMMConcatInit` if your CMM supports the request.

In response to this request code, your CMM should call its `MyCMConcatInit` subroutine. For example, to do this, your CMM may call the Component Manager's `CallComponentFunctionWithStorage` function, passing it the storage handle for this component instance, the component parameters data structure, and the address of your `MyCMConcatInit` subroutine. For a description of the

function prototype to which your subroutine must adhere, see the “ColorSync Manager Reference for Color Management Modules” chapter of the *Advanced Color Imaging Reference* provided on the enclosed CD.

In addition to the storage handle for private storage for this component instance, the `CallComponentFunctionWithStorage` function passes to your `MyCMConcatInit` subroutine a pointer to a data structure of type `CMConcatProfileSet` containing an array of profiles to be used in a subsequent color-matching or color-checking session. The profiles in the array are in processing order—source through destination. The `profileSet` field of the data structure contains the array. If the profile array contains only one profile, that profile is a device-linked profile. For a description of the `CMConcatProfileSet` data structure, see the section “Concatenated Profile Set Structure” in the “ColorSync Manager Reference for Applications and Device Drivers” chapter of the *Advanced Color Imaging Reference* provided on the enclosed CD.

Using the storage passed to your entry point function in the `CMSession` parameter, your `MyCMConcatInit` function should initialize any private data your CMM will need for a subsequent color session involving the set of profiles. Before your function returns control to the Component Manager, your subroutine should store any profile information it requires. In addition to the standard profile information, you should store the profile header’s quality flags setting, the profile size, and the rendering intent. After you return control to the Component Manager, you cannot use the profile references again.

A color-matching or color-checking session for a set of profiles entails various color transformations among devices in a sequence for which your CMM is responsible. Your CMM may use Component Manager functions to call other CMMs if necessary.

There are special guidelines your CMM must follow in using a set of concatenated profiles for subsequent color-matching or gamut-checking sessions. These guidelines are covered in the `MyCMConcatInit` entry in the “ColorSync Manager Reference for Color Management Modules” chapter of the *Advanced Color Imaging Reference*.

Creating a Device-Linked Profile and Opening a Reference to It

When a ColorSync-supportive application or device driver calls the `CWNewLinkProfile` function for your CMM to handle, the Component Manager calls your CMM with the `what` field of the component parameters data structure set to `kCMMNewLinkProfile` if your CMM supports the request.

In response to this request code, your CMM should call its `MyCMNewLinkProfile` subroutine. For example, to do this, your CMM may call the Component Manager's `CallComponentFunctionWithStorage` function, passing it the storage handle for this component instance, the component parameters data structure, and the address of your `MyCMNewLinkProfile` subroutine. For a description of the function prototype to which your subroutine must adhere, see the “ColorSync Manager Reference for Color Management Modules” chapter of the *Advanced Color Imaging Reference* provided on the enclosed CD.

In addition to the storage handle for private storage for this component instance, the `CallComponentFunctionWithStorage` function passes to your `MyCMNewLinkProfile` subroutine a pointer to a data structure of type `CMConcatProfileSet` containing the array of profiles that will make up the device-linked profile.

To handle this request, your subroutine must create a single device-linked profile of type `DeviceLink` that includes the profiles passed to you in the array pointed to by the `profileSet` parameter. Your CMM must create a file specification for the device-linked profile. A device-linked profile cannot be a temporary profile: that is, you cannot specify a location type of `cmNoProfileBase` for a device-linked profile. For information on how to specify the file location, see the “Profile Location Union” and the “Profile Location Structure” in the “ColorSync Manager Reference for Applications and Device Drivers” chapter of the *Advanced Color Imaging Reference* provided on the enclosed CD.

The profiles in the array are in the processing order—source through destination—which you must preserve. After your CMM creates the device-linked profile, it must open a reference to the profile and return the profile reference along with the location specification.

Obtaining PostScript-Related Data From a Profile

There are three very similar PostScript-related request codes that your CMM may support. Each of these codes requests that your CMM obtain or derive information required by a PostScript printer from the specified profile and pass that information to a function supplied by the calling program.

When a ColorSync-supportive application or device driver calls the high-level function corresponding to the request code and your CMM is specified to handle it, the Component Manager calls your CMM with the `what` field of the component parameters data structure set to the corresponding request code if

your CMM supports it. Here are the three high-level functions and their corresponding request codes:

- When the application or device driver calls the `CMGetPS2ColorSpace` function, the Component Manager calls your CMM with a `kCMMGetPS2ColorSpace` request code. To respond to this request, your CMM must obtain the color space data from a source profile and pass the data to a low-level data-transfer function supplied by the calling application or device driver.
- When the application or device driver calls the `CMGetPS2ColorRenderingIntent` function, the Component Manager calls your CMM with a `kCMMGetPS2ColorRenderingIntent` request code. To respond to this request, your CMM must obtain the color rendering intent from the source profile and pass the data to a low-level data-transfer function supplied by the calling application or device driver.
- When the application or device driver calls the `CMGetPS2ColorRendering` function, the Component Manager calls your CMM with a `kCMMGetPS2ColorRendering` request code. To respond to this request, your CMM must obtain the rendering intent from the source profile's header. Then your CMM must obtain or derive the color rendering dictionary for that rendering intent from the destination profile and pass the CRD data to a low-level data-transfer function supplied by the calling application or device driver.

In response to each of these request codes, your CMM should call its subroutine that handles the request. For example, to do this, your CMM may call the Component Manager's `CallComponentFunctionWithStorage` function, passing it the storage handle for this component instance, the component parameters data structure, and the address of your subroutine handler.

For a description of the function prototypes to which your subroutine must adhere for each of these requests, see the appropriate one of the following sections in the “ColorSync Manager Reference for Color Management Modules” chapter of the *Advanced Color Imaging Reference* provided on the enclosed CD:

- For the `kCMMGetPS2ColorSpace` request, see the section `MyCMMGetPS2ColorSpace`.
- For the `kCMMGetPS2ColorRenderingIntent` request, see the section `MyCMMGetPS2ColorRenderingIntent`.
- For the `kCMMGetPS2ColorRendering` request, see the section `MyCMMGetPS2ColorRendering`.

In addition to the storage handle for private storage for this component instance, the `CallComponentFunctionWithStorage` function passes to your subroutine a reference to the source profile containing the data you must obtain or derive, a pointer to the function supplied by the calling program, and a reference constant that you must pass to the supplied function each time your CMM calls it. For `kCMMGetPS2ColorRendering`, your CMM is also passed a reference to the destination profile.

To handle each of these requests, your subroutine must allocate a data buffer in which to pass the particular PostScript-related data to the function supplied by the calling application or driver. Your subroutine must call the supplied function repeatedly until you have passed all the data to it. For a description of the prototype of the application or driver-supplied function, see the `MyColorSyncDataTransfer` entry in the “ColorSync Manager Reference for Applications and Device Drivers” chapter of the *Advanced Color Imaging Reference* provided on the enclosed CD.

For a description of how each of your subroutines must interact with the calling program’s supplied function, see the descriptions of the prototypes for the subroutines in the “ColorSync Manager Reference for Color Management Modules” chapter of the *Advanced Color Imaging Reference* provided on the enclosed CD.

Obtaining the Size of the Color Rendering Dictionary for PostScript Printers

When a ColorSync-supportive application or device driver calls the `CMGetPS2ColorRenderingVMSize` function for your CMM to handle, the Component Manager calls your CMM with the `what` field of the component parameters data structure set to `kCMMGetPS2ColorRenderingVMSize` if your CMM supports the request.

In response to this request code, your CMM should call its `MyCMMGetPS2ColorRenderingVMSize` subroutine. For example, to do this, your CMM may call the Component Manager’s `CallComponentFunctionWithStorage` function, passing it the storage handle for this component instance, the component parameters data structure, and the address of your `MyCMMGetPS2ColorRenderingVMSize` subroutine. For a description of the function prototype to which your subroutine must adhere, see the “ColorSync Manager Reference for Color Management Modules” chapter of the *Advanced Color Imaging Reference* provided on the enclosed CD.

In addition to the storage handle for global data for this component instance, the `CallComponentFunctionWithStorage` function passes to your

Developing Color Management Modules

`MyCMMGetPS2ColorRenderingVMSize` subroutine a reference to the source profile identifying the rendering intent and a reference to the destination profile containing the color rendering dictionary (CRD) for the specified rendering intent.

To handle this request, your CMM must obtain or assess and return the maximum VM size for the CRD of the specified rendering intent.

If the destination profile contains the Apple-defined private tag `'psvm'`, described in the next paragraph, then your CMM may read the tag and return the CRD VM size data supplied by this tag for the specified rendering intent. If the destination profile does not contain this tag, then you must assess the VM size of the CRD.

The `CMPS2CRDVMSizeType` data type defines the Apple-defined `'psvm'` optional tag that a printer profile may contain to identify the maximum VM size of a CRD for different rendering intents.

This tag's element data includes an array containing one entry for each rendering intent and its virtual memory size. For a description of the data structures used to define the tag's element data, see the section "PostScript Color Rendering Dictionary (CRD) Virtual Memory Size Tag Structure" in the "ColorSync Manager Reference for Applications and Device Drivers" chapter of the *Advanced Color Imaging Reference* provided on the enclosed CD.

Flattening a Profile for Embedding in a Graphics File

When a ColorSync-supportive application or device driver calls the `CMFlattenProfile` function for your CMM to handle, the Component Manager calls your CMM with the `what` field of the component parameters data structure set to `kCMMFlattenProfile` if your CMM supports the request.

In response to this request code, your CMM should call its `MyCMMFlattenProfile` subroutine. For example, to do this, your CMM may call the Component Manager's `CallComponentFunctionWithStorage` function, passing it the storage handle for this component instance, the component parameters data structure, and the address of your `MyCMMFlattenProfile` subroutine. For a description of the function prototype to which your subroutine must adhere, see the "ColorSync Manager Reference for Color Management Modules" chapter of the *Advanced Color Imaging Reference* provided on the enclosed CD.

In addition to the storage handle for private storage for this component instance, the `CallComponentFunctionWithStorage` function passes to your `MyCMMFlattenProfile` subroutine a reference to the profile to be flattened, a pointer to a function supplied by the calling program, and a reference constant

your subroutine must pass to the calling program's function when you invoke it.

To handle this request, your subroutine must extract the profile data from the profile, allocate a buffer in which to pass the profile data to the supplied function, and pass the profile data to the function keeping track of the amount of remaining data to be passed.

For a description of the prototype of the function supplied by the calling program, see the entry `MyColorSyncDataTransfer` in the "ColorSync Manager Reference for Applications and Device Drivers" chapter of the *Advanced Color Imaging Reference* provided on the enclosed CD. The "ColorSync Manager Reference for Color Management Modules" chapter also provides details on how your `MyCMMFlattenProfile` subroutine communicates with the function supplied by the calling program.

Unflattening a Profile

When a ColorSync-supportive application or device driver calls the `CMUnflattenProfile` function for your CMM to handle, the Component Manager calls your CMM with the `what` field of the component parameters data structure set to `kCMMUnflattenProfile` if your CMM supports the request.

In response to this request code, your CMM should call its `MyCMMUnflattenProfile` subroutine. For example, to do this, your CMM may call the Component Manager's `CallComponentFunctionWithStorage` function, passing it the storage handle for this component instance, the component parameters data structure, and the address of your `MyCMMUnflattenProfile` subroutine. For a description of the function prototype to which your subroutine must adhere, see the "ColorSync Manager Reference for Color Management Modules" chapter of the *Advanced Color Imaging Reference* provided on the enclosed CD.

In addition to the storage handle for private storage for this component instance, the `CallComponentFunctionWithStorage` function passes to your `MyCMMUnflattenProfile` subroutine a pointer to a function supplied by the calling program and a reference constant that your subroutine must pass to the calling program's function when you invoke it. The calling program's function obtains and returns the profile data to your subroutine.

For a description of the prototype of the function supplied by the calling program, see the `MyColorSyncDataTransfer` entry in the "ColorSync Manager Reference for Applications and Device Drivers" chapter of the *Advanced Color Imaging Reference* provided on the enclosed CD.

Developing Color Management Modules

To handle this request, your subroutine must create a file in which to store the profile data. You should create the file in the temporary items folder. Your `MyCMMUnflattenProfile` subroutine must call the supplied `MyColorSyncDataTransfer` function repeatedly to obtain the profile data. Before calling the `MyColorSyncDataTransfer` function, your `MyCMMUnflattenProfile` function must allocate a buffer to hold the returned profile data.

Your `MyCMMUnflattenProfile` function must identify the profile size and maintain a counter tracking the amount of data transferred to you and the amount of remaining data. This information allows you to determine when to call the `MyColorSyncDataTransfer` function for the final time.

For a description of the prototype of the function supplied by the calling program, see the entry `MyColorSyncDataTransfer` in the “ColorSync Manager Reference for Applications and Device Drivers” chapter of the *Advanced Color Imaging Reference* provided on the enclosed CD. The “ColorSync Manager Reference for Color Management Modules” chapter in the same book also provides details on how your `MyCMMUnflattenProfile` subroutine communicates with the function supplied by the calling program.

Summary of the Color Management Modules

Constants

```
enum {
    CMMInterfaceVersion= 1
};

/* request codes (required) */
enum {
    kCMMInit                = 0,
    kCMMMatchColors         = 1,
    kCMMCheckColors         = 2,
    kNCMMInit               = 6,
};

/* request codes (optional) */
enum {
    kCMMValidateProfile      = 8,
    kCMMFlattenProfile       = 14,
    kCMMUnflattenProfile     = 15,
    kCMMMatchBitmap          = 9,
    kCMMCheckBitmap          = 10,
    kCMMMatchPixMap          = 3,
    kCMMCheckPixMap          = 4,
    kCMMConcatenateProfiles  = 5,
    kCMMConcatInit          = 7,
    kCMMNewLinkProfile       = 16,
    kCMMGetPS2ColorSpace     = 11,
    kCMMGetPS2ColorRenderingIntent = 12,
    kCMMGetPS2ColorRendering = 13,
    kCMMGetPS2ColorRenderingVMSize = 17
};
```

Functions

Required Functions

pascal CError MyNCMInit	(ComponentInstance CMSession, CMProfileRef srcProfile, CMProfileRef dstProfile);
pascal CError MyCMMatchColors	(ComponentInstance CMSession, CMColor *myColors, unsigned long count);
pascal CError MyCMCheckColors	(ComponentInstance CMSession, CMColor *myColors, unsigned long count, long *result);
pascal CError CMInit	(ComponentInstance CMSession, CMProfileHandle srcProfile, CMProfileHandle dstProfile)

Optional Functions

pascal CError MyCMValidateProfile (ComponentInstance CMSession, CMProfileRef prof, Boolean *valid);
pascal CError MyCMMatchBitmap	(ComponentInstance CMSession, const CMBitmap *bitmap, CMBitmapCallbackUPP progressProc, void *refCon, CMBitmap *matchedBitmap);
pascal CError MyCMCheckBitmap	(ComponentInstance CMSession, const CMBitmap *bitmap, CMBitmapCallbackUPP progressProc, void *refCon, CMBitmap *resultBitmap);
pascal CError MyCMConcatInit	(ComponentInstance CMSession, CMConcatProfileSet *profileSet);
pascal CError MyCMMatchPixMap	(ComponentInstance CMSession, PixMap *myPixMap, CMBitmapCallbackUPP progressProc, void *refCon);
pascal CError MyCMCheckPixMap	(ComponentInstance CMSession, const PixMap *myPixMap, CMBitmapCallbackUPP progressProc, BitMap *myBitMap, void *refCon);
pascal CError MyCMNewLinkProfile	(ComponentInstance CMSession, CMProfileRef *prof, const CMProfileLocation *targetLocation, CMConcatProfileSet *profileSet);


```

pascal CLError MyCMConcatenateProfiles (
    ComponentInstance CMSession, CMProfileHandle thru,
    CMProfileHandle dst,
    CMProfileHandle *newDst);

pascal CLError MyCMMGetPS2ColorSpace (
    ComponentInstance CMSession, CMProfileRef srcProf,
    unsigned long flags, CMFlattenUPP proc,
    void *refCon);

pascal CLError MyCMMGetPS2ColorRenderingIntent (
    ComponentInstance CMSession, CMProfileRef srcProf,
    unsigned long flags, CMFlattenUPP proc,
    void *refCon);

pascal CLError MyCMMGetPS2ColorRendering (
    ComponentInstance CMSession, CMProfileRef srcProf,
    CMProfileRef dstProf, unsigned long flags,
    CMFlattenUPP proc, void *refCon);

pascal CLError MyCMMGetPS2ColorRenderingVMSize (
    ComponentInstance CMSession, CMProfileRef srcProf,
    CMProfileRef dstProf, unsigned long vmSize);

pascal CLError MyCMMFlattenProfile (
    ComponentInstance CMSession, CMProfileRef prof,
    unsigned long flags, CMFlattenUPP
    proc, void *refCon);

pascal CLError MyCMMUnflattenProfile (
    ComponentInstance CMSession,
    FSSpec *resultFileSpec,
    CMFlattenUPP proc, void *refCon);

```


Developing ColorSync-Supportive Device Drivers

Contents

About ColorSync-Supportive Device Driver Development	6-4
The ColorSync Manager Programming Interface	6-4
Devices and Their Profiles	6-5
The Profile Format and Its Cross-Platform Use	6-6
Storing and Handling Device Profiles	6-6
How You Use Profiles	6-7
Devices and Color Management Modules	6-8
Providing ColorSync-Supportive Device Drivers	6-8
Providing Minimum Support	6-9
Providing More Extensive ColorSync Support	6-9
Developing Your ColorSync Supportive Device Driver	6-10
Determining If the ColorSync Manager Is Available	6-11
Interacting With the User	6-11
Searching for Profiles and Displaying Their Names to the User	6-12
Setting the Rendering Intent Selected by the User	6-15
Setting the Color-Matching Quality Selected by the User	6-17
Color Matching an Image to Be Printed	6-22

This chapter describes how you can use the ColorSync Manager to provide ColorSync-supportive device drivers for peripherals. It first describes the three classes of devices—input, display, and output—that the ColorSync Manager supports.

This chapter describes what you must do to provide minimum ColorSync support. Then, using a QuickDraw-based printer device driver as an example, it describes some of the color-matching features that a device driver can provide, focusing on features offered through a user interface.

Although the features described in this chapter are commonly provided by printer device drivers, they can also be provided by other applications that support the ColorSync Manager. For this reason, features described in this chapter are also addressed in the chapter “Developing ColorSync-Supportive Applications,” which you should read in addition to this chapter. “Developing ColorSync-Supportive Applications” includes hints that will benefit your development of a ColorSync-supportive device driver and code samples that illustrate approaches you can take in implementing processes, such as extracting an embedded profile, that you may want your driver to perform.

You need to read this chapter if your device driver for an input, display, or output device will support the ColorSync Manager and allow users of the peripheral supported by the driver to produce images that can be color matched or that are color matched.

Before you read this chapter, you should read the chapter “Introduction to the ColorSync Manager.” It explains color theory and color management systems (CMSs), and it provides an overview of the ColorSync Manager CMS, including the use of profiles. It also explains key terms used throughout this chapter but not defined again in it.

While reading this chapter, you might want to refer to the chapter “ColorSync Manager Reference for Applications and Device Drivers” in the *Advanced Color Imaging Reference* for details related to functions this chapter discusses.

In addition to one or more profiles for the device, device drivers that support the ColorSync Manager can provide their own color management module (CMM) created to perform the best possible color matching for the device the driver supports. Peripherals manufacturers can obtain CMMs and profiles from vendors who create them, or they can create their own profiles and CMMs. For a list of profile vendors, contact Apple Computer’s Developer Technical Support organization.

If you are creating your own CMM, you should also read the chapter “Developing Color Management Modules” in this book and the “ColorSync Manager Reference for Color Management Modules” in the *Advanced Color Imaging Reference*.

IMPORTANT

This chapter does not include a ColorSync Manager quick reference section. Instead, the complete ColorSync Manager quick reference, including enumerations, data structures, functions, and result codes for applications and device drivers, is provided at the end of the chapter “Developing ColorSync-Supportive Applications.” ▲

About ColorSync-Supportive Device Driver Development

Your device driver can provide minimal ColorSync Manager support, or it can make extensive use of the ColorSync Manager functions to provide your users with full color-matching capability. This section describes some aspects to consider in regard to profiles you provide for your device.

The ColorSync Manager requires Color QuickDraw and System 7.0 or later. The Component Manager is packaged with the ColorSync Manager and installed at startup in systems that do not include it. Your device driver must include one or more profiles for the device you support and, optionally, a CMM if you want to use a custom one rather than the Apple-supplied default CMM.

The ColorSync Manager Programming Interface

Your device driver calls the functions defined by the ColorSync Manager programming interfaces to handle such tasks as color matching, color conversion, profile management, extracting profiles embedded in a document to be printed, profile searching and accessing, and providing color-rendering information to PostScript printers that perform color matching. The ColorSync Manager includes the following five interface files for 68K and PowerPC development:

`CMApplication.h`

Interface to the ColorSync Manager functions and data types for applications and device drivers.

CMConversions.h

Interface for base-derived color space conversions.

CMICCPProfile.h

Definitions for the version 2.0 profile for profile developers. This interface file contains enumerations and structures, such as the version 2.0 profile header, that your application requires. Therefore, you must include it.

CMMComponent.h

Interface to the functions and data types for CMMs.

CMPRComponent.h

Interface for Profile Responder components for ColorSync 1.0 backward-compatibility support. If your application provides backward compatibility, you must include this interface file.

Devices and Their Profiles

To assess the way each device interprets color, color scientists and profile developers perform device characterizations. This process, which entails measuring the gamut of a device, yields a color profile for that device. Device profiles are of paramount importance to any color management system (CMS) because they characterize the unique color behavior of each device and allow color matching to occur. Device profiles are used by CMMs that perform the low-level calculations required to match colors from a source device to a destination device.

To support the ICC specification, the ColorSync Manager supports the following three classes of devices for which you can provide device drivers:

- an input device, such as a scanner or a digital camera
- a display device, such as a monitor or a liquid crystal display
- an output device, such as a printer

For each class of device, the ICC defines a device profile type, each with its own signature. Here are the profile type signatures:

'scnr'	Designates an input device such as a scanner or a digital camera.
'mntr'	Designates a display device such as a monitor or a liquid crystal display (LCD).
'prtr'	Designates an output device such as a printer.

A profile creator specifies the signature denoting the profile type in the profile header's `profileClass` field.

Whether you create a profile for your device or obtain one from a profile vendor, your device driver must provide at least one profile for its device. However, you can provide more than one profile for the same device to characterize it differently. Although a printer that your device driver supports may have a number of profiles for different conditions, such as the use of foils or different grades of paper, all of its profiles will use the same 'prtr' profile signature.

The Profile Format and Its Cross-Platform Use

Device profiles follow the ICC profile format, an industry standard. You can provide a single profile or a set of profiles that can be used across different operating systems for the device your driver supports. The common profile format specified by the ICC allows end users to transparently move profiles and images with embedded profiles between different operating systems.

The profile structure is defined as a header followed by a tag table which, in turn, is followed by a series of tagged elements that your device driver can access randomly and individually. Using the ColorSync Manager functions, you can obtain the profile header to read and modify its contents and you can get and set individual tags and their element data.

This profile structure is referred to by the ICC as version 2.0. Version 2.0 profiles require more information and are larger than ColorSync 1.0 profiles, which were originally memory based. Because version 2.0 profiles are larger, they are disk-based. The ICC profile format specification defines how version 2.0 profiles can be stored as disk files and how profiles can be embedded in common graphics file formats such as PICT and TIFF. The ColorSync Manager provides a data structure that you can use to identify the location of a profile. It also provides functions you can use to embed a profile in or extract it from a PICT file.

Storing and Handling Device Profiles

Device profiles reside in files in the ColorSync™ Profiles folder (within the Preferences folder of the System Folder), in pictures, or with device drivers. Files that contain profiles keep them in the data fork and are of type 'prof'.

By convention, profiles not embedded in documents containing the images they are associated with are stored in the ColorSync™ Profiles folder. Although

you can decide where to store your profiles, to make them available to other applications you should store them in the ColorSync™ Profiles folder. Applications that perform soft proofing or gamut checking can search the folder for specific types of profiles in order to provide a selection menu or list to the user. If your profiles are not available, applications will not be able to offer use of them to their users for color matching or gamut checking. These applications will not be able to color match to your device unless they provide a profile to be used for it.

The ColorSync™ Profiles folder may contain profiles for your device provided by applications for special purposes. For this reason, when your device driver itself displays a selection menu or list to the user, you should search not only your private profile location storage, if you use one, but also the ColorSync™ Profiles folder to make sure that you offer your users a complete list of available profiles for your device.

The ColorSync™ Profiles folder can contain both ColorSync 1.0 profiles and version 2.0 profiles. However, your device driver will be able to search for only version 2.0 profiles. This is because the ColorSync Manager 2.0 search functions that you use to look for profiles in the folder do not acknowledge ColorSync 1.0 profiles.

How You Use Profiles

For most of the ColorSync Manager functions that your device driver calls, you will need to supply references to profiles for both the source device on which the image was created and the destination device for which it is to be color matched and where it will be rendered.

The driver for an input device such as a scanner typically embeds the scanner profile used to create the image in the document containing the image. The driver for a device that displays an existing image on the system's display or a printer device that prints a color-matched image typically extracts the embedded profile that accompanies the image from the document containing the image.

Images created using input devices are commonly color matched using the profile for the input device as the source profile and the system profile for the display as the destination profile. Images that are created, depicted, or modified on a display device and that are destined for an output device such as a printer are color matched using the profile for the display as the source profile and the printer's profile as the destination one.

To use a profile, you must first obtain a reference to the profile. For information on how to obtain a profile reference, see the chapter “Developing ColorSync-Supportive Applications.”

Devices and Color Management Modules

Your device driver can use the color conversion functions, described in the “ColorSync Manager Reference for Applications and Device Drivers” chapter of the *Advanced Color Imaging Reference*, to convert colors between color spaces belonging to the same base family without relying on a CMM. However, color matching, gamut checking, providing color rendering dictionaries to PostScript printers, and other tasks you perform using the ColorSync Manager functions all require use of a CMM. It is the CMM that actually carries out the work of the ColorSync Manager functions, for example, performing the low-level calculations required to match colors from a source device to a destination device.

If your ColorSync-supportive device driver can use the Apple-supplied default CMM, you only need to provide one or more profiles for your device. However, you may want to provide a custom CMM that is optimized for your device and its profiles. For example, a profile can provide private tags containing information a custom CMM might use to achieve better results for the device.

If you provide your own CMM, you can create it or obtain one from a vendor. For information describing how to create a CMM, see the chapter “Developing Color Management Modules” in this book and the chapter “ColorSync Manager Reference for Color Management Modules” in the *Advanced Color Imaging Reference*.

Providing ColorSync-Supportive Device Drivers

Your ColorSync-supportive device driver can provide your users with various features based on color matching, depending on the type of device you support. This section describes what you should do to provide minimum ColorSync Manager support. Then, using a color printer as an example, this section lists some of the features you can implement to provide more extensive support.

Providing Minimum Support

The minimum level of ColorSync Manager support you should provide differs depending on the type of device your driver supports.

For a scanner, you should embed the scanner profile used to create the image in the document containing the image; this is also referred to as *tagging* an image. If you do not tag the image with the profile, you should at least make the profile for the image available so that it can be used for color matching. If you do not provide the scanner profile, an application or driver that attempts to color match the scanned image will use the system profile as the source profile and produce results inconsistent with the colors of the original image.

For a display device driver or a printer device driver, you must preserve images tagged with a profile by not stripping out picture comments used to embed profiles or by leaving profiles in documents that use other methods to include them. For example, if your driver displays or prints PICT files but does not perform color matching, your driver should not strip out the ColorSync-related picture comments that are used to embed profiles in PICT files, begin and end use of a specific profile, and enable and disable color matching. Even though your driver may not make use of the profiles, another display or printer driver or an application may.

If you don't perform color matching but you want to allow applications that do to produce images that are color matched for your device, you should provide a device profile to be used as the destination profile. If you provide a profile for your display or printer and place it in the ColorSync™ Profiles folder, applications that perform color matching can use it to create a color-matched image expressed in the colors of your device's gamut. A user can then print a color-matched image using the printer your driver supports.

Providing More Extensive ColorSync Support

Instead of relying on an application to color match an image for your printer, your printer driver can color match the image itself before sending it to the printer. To perform color matching, your printer driver must obtain a reference to the source profile. Documents containing images to be printed often contain an embedded profile along with the image. To use the source profile, your printer driver must be able to extract it. If an image is not accompanied by a source profile, the system profile is used. In this case, your driver should provide an interface that allows the user to select the rendering intent to be used.

You can provide an interface that offers your user additional features. Your interface can

- allow your user to turn ColorSync Manager color matching on or off before printing.
- offer your user selection menus, allowing the user to choose
 - the rendering method to be used in color matching the image (perceptual, colorimetric, or saturation)
 - the color-image quality (normal, draft, or best)

Allowing your user to turn color matching on or off

If an application that creates or modifies an image already performed color matching using your printer profile as the destination profile, your user could turn off color matching. To provide this capability, your driver should support the `PrGeneral` function with the `enableColorMatchingOp` operation code. For information on the `PrGeneral` function, see *Inside Macintosh: Imaging With QuickDraw*. The `enableColorMatchingOp` operation code constant is defined by the ColorSync Manager. ♦

Some of these features are discussed later in this chapter and in the chapter “Developing ColorSync-Supportive Applications.”

Developing Your ColorSync Supportive Device Driver

Your device driver can implement certain features using the ColorSync Manager to support color matching and its associated processes. This section describes how to tell whether the ColorSync Manager is installed on the system running your device driver.

Then it describes how to implement features and provide a user interface that offers a user control over the settings for these features. You can provide a user interface that allows the user to choose which profile to use for the device your driver supports. The user interface can also allow the user to choose the color-rendering intent and the color-matching quality. Finally, this section describes how to color match images sent to your device driver before sending

them to your printer. Many of the tasks that your device driver performs to support the ColorSync Manager can also be performed by other kinds of color-matching applications. These tasks are mentioned in this chapter, but not explained in detail. For details, refer to the chapter “Developing ColorSync-Supportive Applications.”

Determining If the ColorSync Manager Is Available

To determine if the ColorSync Manager (version 2.0) is available on a 68K-based Mac, use the `Gestalt` function with the `gestaltColorMatchingVersion` selector. This function returns the result `gestaltColorSync20` if the ColorSync Manager is available.

To determine if the ColorSync Manager shared libraries have been loaded on a Power Macintosh, use the `Gestalt` function with the `gestaltColorMatchingAttr` selector. Test the bit field (bit 1) indicated by the `gestaltColorMatchingLibLoaded` constant in the response parameter. If the bit is set, the ColorSync Manager shared libraries are loaded.

Interacting With the User

Using selection menus, lists, and dialog boxes, you can provide your user with choices that influence the color-matching process. For example, you can offer the user

- a list of profiles to select from. You can allow the user to choose the appropriate profile for your printer in its current state. To provide a list of profiles for the user to select from, you must first search for the relevant profiles. The next section outlines approaches you can take.
- a selection menu or dialog box for specifying how the image will be color matched. If the source profile is embedded with the image, the source profile specifies the rendering intent to be used. However, if the source profile is not provided and the system profile is used as the source profile, you should allow the user to select the rendering intent to be used. Your user has different requirements for different kinds of images, such as scanned photographs, posters, pie charts, and book illustrations. You can allow users to specify their intentions for the process to be used in matching the source image's colors when they fall out of gamut for your printer. After the user chooses a rendering intent, you can use the selection to set the source

profile's header. "Setting the Rendering Intent Selected by the User" on page 6-15 explains this process.

- a selection menu or dialog box for choosing which color-matching quality of image to produce. A user may want to produce a draft of the image quickly for review before producing the best possible quality of the image. After the user chooses a color-matching quality, you can use the selection to set the source profile's header. "Setting the Color-Matching Quality Selected by the User" on page 6-17 explains how to do this.

Searching for Profiles and Displaying Their Names to the User

The ColorSync Manager search functions let you identify types of profiles or profiles having certain characteristics. You can use these functions to search through available version 2.0 profiles in the ColorSync™ Profiles folder and hone in on profiles that meet criteria you specify. You can set up a search to look for certain classes of device profiles for a specific manufacturer's device, such as printer profiles designed for the printer model your device driver supports. You can even refine the search to look for a specific printer profile for the printer's current state, for example, a profile especially designed for your printer's use of foils and a certain type of ink.

IMPORTANT

The ColorSync Manager 2.0 search functions can identify only version 2.0 profiles in the ColorSync™ Profiles folder. If the folder contains ColorSync 1.0 profiles, they are not investigated during the search. ▲

Searching for profiles entails the following four steps:

1. Defining the search criteria

You can define the search criteria broadly based on the profile's device type only or refine the search based on other specific characteristics of a profile carried in its header or elements. The section "Defining Search Criteria" describes this step.

2. Running the search

After you set up the search criteria, you must call the `CMNewProfileSearch` function to start the search and find all profiles that match your requirements. "Running the Search" on page 6-14 explains how to do this.

3. Gaining access to the search results

Using the search results, you can now obtain references to each of the profiles that match your criteria by calling the `CMSearchGetIndProfile` function. “Obtaining References to the Qualifying Profiles” on page 6-14 describes how to do this.

4. Getting the names of qualifying profiles to display to your user

Once you have references to each of the qualifying profiles, you can obtain the names of the profiles to present to your user. “Getting the Names of Qualifying Profiles” on page 6-15 explains this step.

Defining Search Criteria

Before you begin a profile search, you must describe aspects of the profiles you are looking for. To help you, the ColorSync Manager provides a data structure of type `CMSearchRecord` that you use to identify the search criteria. Your device driver can declare an instance of the search record and assign values to its fields to characterize the profiles you want to find. The search record data structure of type `CMSearchRecord` is described in the “ColorSync Manager Reference for Applications and Device Drivers” chapter of the *Advanced Color Imaging Reference*.

You define the parameters of a search by first setting the fields of the search record whose values you want matched and then setting the `searchMask` field to indicate the operative fields. If you don’t set the field mask, your search criteria are ignored.

To create a refined search that seeks all profiles belonging to a certain class that also meet other criteria, you must set multiple fields of the search record. For example, suppose you want to find all printer profiles for your manufacturer’s printer and you also want to filter out profiles unless they are for a certain device model. To do this, you need to set the search record’s `deviceManufacturer` field to the signature that identifies the manufacturer. This must be the same signature specified in the profile header’s `deviceManufacturer` field when the manufacturer or vendor created the profile. You must also set the `deviceModel` field to the value that identifies the printer model whose profiles you want to identify. After you set these fields, you must set the `searchMask` to identify the operative fields. To do this, you can use the constants `cmMatchProfileClass`, `cmMatchManufacturer`, and `cmMatchModel`. After you define a search record that establishes your criteria, you must initiate the search.

For a description of the enumeration that defines constants for profile class signatures, see the section “Profile Classes” in the chapter “ColorSync Manager Reference for Applications and Device Drivers” in the *Advanced Color Imaging Reference* on the enclosed CD. For a description of the enumeration that defines constants for the search mask, see the section “Profile Search Record” in the same chapter.

Running the Search

Passing to this function the search record whose fields you set in the previous step, you call the `CMNewProfileSearch` function to run the search on all profiles in the ColorSync™ Profiles folder. The `CMNewProfileSearch` function searches the ColorSync™ Profiles folder for version 2.0 profiles that meet your criteria. In response, the function returns to your driver a reference to a search results private data structure containing a list of all profiles that match your description.

The function also returns a one-based count telling you how many profiles are in the list. You use this count, along with the search result reference, to navigate the list in order to identify a specific profile and obtain information about it. Your device driver cannot gain direct access to the contents of the search result list.

Obtaining References to the Qualifying Profiles

Using the results of the search, you can now obtain references to each of the qualifying profiles. The one-based count returned by the `CMNewProfileSearch` function identifies the number of profiles in the list. You can use this count to set the bounds of a loop you can define to obtain a profile reference for each qualifying profile. Identifying the index position of a profile in the search result list, you can call the `CMSearchGetIndProfile` function repeatedly, incrementing through the list until you obtain references to all of its profiles. The sample code listing in “Searching for Profiles in the ColorSync™ Profiles Folder” of the “Developing ColorSync-Supportive Applications” chapter shows one approach you can take.

The ColorSync Manager preserves the results of a search you perform until you discard the private data structure containing the search result list by calling `CMDisposeProfileSearch`. If you know that the user has updated the ColorSync™ Profiles folder since you searched it, you can call the `CMUpdateProfileSearch` function to update the search result without providing the search specification again.

Getting the Names of Qualifying Profiles

After you obtain references to the profiles in the search result list, you must get the names of the profiles so that you can display them in the selection list your user interface provides. You can get the profile names either within your profile reference loop or outside it.

When you are finished with the profiles, you must call the `CMCloseProfile` function for each one to dispose of the references and release the memory they use.

Setting the Rendering Intent Selected by the User

The ColorSync Manager offers four rendering intents—perceptual, relative colorimetric, saturated, and absolute colorimetric. Every profile supports these. The first three are commonly used to render images matching the colors of a source image to the color gamut of the destination device on which the resulting image is to be rendered in the most optimum way for the type of image.

If the source profile is embedded with the image, the source profile specifies the rendering intent to be used. However, if the source profile is not provided and the system profile is used as the source profile, you should allow the user to select the rendering intent to be used.

To allow users to choose the rendering intent most appropriate for color matching their graphical image, you can provide a menu or a dialog box identifying the rendering intent options available. To help your user in choosing the appropriate intent, you can provide meaningful information that identifies the best use of each intent. Users can then select the rendering intent that best maintains important aspects of the image.

Instead of simply listing the available rendering intents by the technical names used to refer to them, you can indicate how they are best used, basing your presentation on this background information:

- For *perceptual matching*, all the colors of a given gamut may be scaled to fit within another gamut. This intent is the best choice for realistic images, such as scanned photographs.
- For *saturation matching*, the relative saturation of colors is maintained from gamut to gamut. Rendering the image using this intent gives the strongest

colors and is the best choice for bar graphs and pie charts, in which the actual color displayed is less important than its vividness.

- For *relative colorimetric matching*, the colors that fall within the gamuts of both devices are left unchanged. Some colors in both images will be exactly the same, a useful outcome when colors must match quantitatively. This intent is best suited for logos.

After the user selects the intent to be used, you must modify the `renderingIntent` field of the system profile's header to reflect the choice.

To put the rendering intent chosen by the user in the profile header, follow these steps:

- 1. Obtain a profile reference to the system profile.**

"Identifying the Current System Profile" in the chapter "Developing ColorSync-Supportive Applications" describes how to do this.

- 2. Get the profile header of the system profile.**

Passing the profile reference to the function, you can use the `CMGetProfileHeader` function to obtain the profile's header. The function returns the profile header using a union of type `CMAppleProfileHeader`. You can use this function for both ColorSync 1.0 profiles and version 2.0 profiles. For a version 2.0 profile, you use the `CM2Header` data structure. For a version 1.0 profile, you use the `CMHeader` data structure. For a description of the profile headers, see the chapter "ColorSync Manager Reference for Applications and Device Drivers" in the *Advanced Color Imaging Reference* on the enclosed CD. This book also describes the `CMGetProfileHeader` function.

- 3. Assign the new rendering intent to the header field.**

To assign a rendering intent to the system profile header's `renderingIntent` field, use the constants defined by the following enumeration:

```
enum {
    cmPerceptual      = 0,
    cmRelativeColorimetric= 1,
    cmSaturation      = 2,
    cmAbsoluteColorimetric= 3
};
```

4. Set the modified profile header of the system profile.

After you assign the rendering intent, you must replace the header by calling the `CMSetProfileHeader` function. You can use the `CMSetProfileHeader` function to set a header for a version 1.0 or a version 2.0 ColorSync profile. You pass the header you supply in the `CMAppleProfileHeader` union, which is described in the chapter “ColorSync Manager Reference for Applications and Device Drivers” in the *Advanced Color Imaging Reference* on the enclosed CD.

You can now use the system profile to create a color world for the color-matching process. For information on how to create a color world, see the chapter “Developing ColorSync-Supportive Applications.”

The profile header is temporarily modified and the rendering intent change is discarded when you call the `CMCloseProfile` function. To preserve the change, you must call the `CMUpdateProfile` function.

Listing 6-1 on page 6-20 shows how to set the rendering intent for a profile.

Setting the Color-Matching Quality Selected by the User

The ColorSync Manager provides a feature, called the *quality flags settings*, that controls the quality of the color-matching process in relation to the time required to perform the match. This feature, which is not a standard feature defined by the ICC profile format specification, works by letting you manipulate certain bits of the profile header’s `flags` field. There are three quality flag settings: normal, draft, and best. For a description of the profile header’s `flags` field, see the chapter “ColorSync Manager Reference for Applications and Device Drivers” in the *Advanced Color Imaging Reference*.

Normal mode is the default setting. Color matching using draft mode takes the least time and produces the least exact results. Color matching using best mode takes the longest time but produces the finest results.

Users sometimes want to produce review drafts of images quickly before expending the time to produce the best-quality final copy. Your interface can allow them this flexibility by offering a selection menu or dialog box that provides the three options.

After the user selects the color-matching quality, you can use the selection to set the appropriate bits of the source profile's `flags` field. To set the color-matching quality chosen by the user, follow these steps:

1. Obtain a profile reference to the source profile.

“Obtaining Profile References” in the chapter “Developing ColorSync-Supportive Applications” describes how to do this.

2. Get the profile header of the source profile.

Passing the profile reference to the function, you can use the `CMGetProfileHeader` function to obtain the profile's header. The function returns the profile header using a union of type `CMAppleProfileHeader`.

3. Optionally, test the current setting of the source profile header's flags.

The `flags` field of the source profile header is a long word coded in big-endian notation. Big-endian notation is a means of encoding data in which the first byte within the 16-bit and 32-bit quantities is the most significant. The ICC profile consortium reserves the first 2 bits of the low word for its own use. The least significant 2 bits of the high word constitute the quality flag settings used to specify the quality for the color matching. To evaluate and interpret the current setting of the quality flags bits, you can take these steps, in order:

- ☐ Right-shift the 16 bits.
- ☐ Mask off the high 14 bits.
- ☐ Compare the result with values defined by the following enumeration:

```
enum {
    cmNormalMode    = 0, cmDraftMode    = 1, cmBestMode    = 2
};
```

4. Set the quality flags bits to the value your user selected.

To set the quality flag, you can use the constants defined by the enumeration provided by the ColorSync Manager and shown in step 3.

5. Set the source profile with the modified profile header.

After you set the `flags` field based on the user's selection, you must replace the header by calling the `CMSetProfileHeader` function. You use the `CMSetProfileHeader` function to set the header. You pass the header you

supply in the `CMAppleProfileHeader` union, which is described in the chapter “ColorSync Manager Reference for Applications and Device Drivers” in the *Advanced Color Imaging Reference* on the enclosed CD.

You can now use the source profile to create a color world for the color-matching process. For information on how to create a color world, see the chapter “Developing ColorSync-Supportive Applications.”

The profile header is temporarily modified and the `flags` field change is discarded when you call the `CMCloseProfile` function. To preserve the change, you must call the `CMUpdateProfile` function.

Listing 6-1 shows how to set the system profile’s quality flag to best mode for producing the highest-quality color-matched image, and shows how to set the rendering intent to saturation before setting up a color world based on the modified system profile and the printer profile.

Listing 6-1’s `MySetHeader` function initializes the `CMProfileRef` data structures it will use for the system profile and the printer profile before it calls the following two functions—the ColorSync Manager `CMGetSystemProfile` function to obtain a reference to the system profile and its own `MyGetPrinterProfile` function to obtain a reference to the profile for its printer.

The source profile—in this case, the system profile—not the printer profile, determines the quality mode and the rendering intent to be used in color matching the image to the destination printer. Now that it has a reference to the system profile, the code can obtain the profile’s header. It does this by calling the `CMGetProfileHeader` function, specifying the reference it obtained to the system profile.

Using the `kSpeedAndQualityFlagMask` constant it defined earlier, the code clears the quality mode bits of the system profile’s `flags` field. Then it sets the quality mode bits to `cmBestMode` to specify best mode quality for color matching. The least significant 2 bits of the `flags` field’s high word constitute the quality flag. After setting the quality flag, the code sets the system profile header’s `renderingIntent` field to `cmSaturation`.

Now that the code has modified the system profile’s header to indicate the user’s selections, it calls the `CMSetProfileHeader` function to write the profile header to the profile. Because the driver code intends to use the values the user selected only to color match the image in the user’s document to be printed, it does not permanently preserve the header field changes. When the code closes its reference to the system profile after having built the color world, the system profile’s header modifications are discarded. To write the changes to the profile to preserve them, the code must include a call to the `CMUpdateProfile` function.

Using the temporarily modified system profile, the code calls the `NCWNewColorWorld` function to create the color world, closes its references to both the system and printer profiles, and color matches the image before sending it to the printer. When it no longer needs the color world, the code calls the `CWDisposeColorWorld` function to close the color world and release the memory it uses. Finally, the code tests to ensure that the profile references are closed.

Listing 6-1 Modifying the system profile header's quality flag and setting the header

```
#include <Types.h>
#include <CMApplication.h>
#include <stdio.h>
#include <assert.h>

#define kMajorVersionMask    0xFF000000

void MySetHeader(void);

CMError MyGetPrinterProfile(CMProfileRef* printerProf);

/* for CM2Header.profileVersion */
#define kMajorVersionMask 0xFF000000

/* two bits used to specify speed & quality; must be shifted left 16 bits in
   flag's long word */
#define kSpeedAndQualityFlagMask 0X00000003

void MySetHeader(void)
{
    CMError          cmErr;
    CMProfileRef     sysProf;
    CMAppleProfileHeader sysHeader;
    CMProfileRef     printerProf;
    CMWorldRef       cw;

    sysProf = NULL;
    printerProf = NULL;
    cw = NULL;
```

```

cmErr = CMGetSystemProfile(&sysProf);
if (cmErr == noErr)
{
    cmErr = MyGetPrinterProfile(&printerProf);
}

if (cmErr == noErr)
{
    cmErr = CMGetProfileHeader(sysProf, &sysHeader);
}

if (cmErr == noErr)
{
    sysHeader.cm2.flags&= ~(kSpeedAndQualityFlagMask << 16);
    sysHeader.cm2.flags|= (cmBestMode << 16);

    sysHeader.cm2.renderingIntent = cmSaturation;

    cmErr = CMSetProfileHeader(sysProf, &sysHeader);
}
if (cmErr == noErr)
{
    cmErr = NCWNewColorWorld(&cw, sysProf, printerProf);

    (void) MyCMCloseProfile(sysProf);
    sysProf = NULL;
    (void) CMCloseProfile(printerProf);
    printerProf = NULL;
}

    .
    .
    .

/* device-driver functions that use the color world to color match
   the image and send it to the printer belong here */

    .
    .
    .

```

```

if (cw != NULL)
{
    CWDiscardColorWorld(cw);
}

/* close open profiles in case of error */
if (sysProf != NULL)
{
    (void) CMCloseProfile(sysProf);
}
if (printerProf != NULL)
{
    (void) CMCloseProfile(printerProf);
}
}

```

Color Matching an Image to Be Printed

The ColorSync Manager provides high-level and low-level color-matching functions. Printer device drivers usually perform color matching using the low-level ColorSync Manager functions to match all QuickDraw operations as they pass through the bottleneck routines.

When the stream of QuickDraw data sent to your printer device driver contains a profile embedded using picture comments, your driver should extract the embedded profile using the ColorSync Manager's `CMUnflattenProfile` function. After you extract the profile and open a reference to it, you should create a new color world based on the extracted profile and a profile for your printer. For information on how to extract an embedded profile, see the chapter "Developing ColorSync-Supportive Applications." This chapter also describes how to create a color world.

If the QuickDraw data stream does not contain embedded profiles, your driver should use the system profile as the source profile in creating the color world.

You should then match subsequent QuickDraw operations using the color world before sending them to your printer.

Color Manager

Contents

About the Color Manager	7-3
Graphics Devices	7-4
Color Tables	7-5
Inverse Tables	7-6
Inverse Tables in Action	7-10
Hidden Colors	7-11
Building Inverse Tables	7-12
Using the Color Manager	7-13
Customizing Search Functions	7-13
Customizing Complement Functions	7-15
Managing the Device CLUT	7-16
Summary of the Color Manager	7-19
Constants and Data Types	7-19
Color Manager Functions	7-20
Application-Defined Functions	7-21

The **Color Manager** assists Color QuickDraw in mapping your application's color requests to the actual colors available. Most applications never need to call the Color Manager directly. The material in this chapter is provided for specialized applications that require a color-mapping method other than the one used by the Color Manager.

You need to understand Color QuickDraw concepts, terminology, and data structures when using the material in this chapter. You should be familiar with RGB color values, color tables, pixel maps, and graphics devices, as described in *Inside Macintosh: Imaging With QuickDraw*.

Color Manager functions are the intermediary between such high-level software as Color QuickDraw, the Palette Manager, and the Color Picker Manager, and the lower-level of video cards and screens. The vast majority of applications never need to use Color Manager functions directly.

About the Color Manager

The Color Manager is optimized to work with graphics hardware that contains a color lookup table (CLUT), a data structure that maps the index values into actual colors. Color QuickDraw supports two kinds of devices:

- Indexed devices, which contain hardware that converts a **pixel value** stored in the card's video RAM to some actual color. The pixel value can be an index to any of the colors in the CLUT for the device, and for most CLUTs the set of colors can be changed.
- Direct devices, which display the RGB color values stored in the card's video RAM. Unlike indexed devices, values placed in the frame buffer of direct devices produce the same color every time.

Color QuickDraw uses the Color Manager to determine which entry in a device's CLUT best maps to a color. When your application uses the `RGBForeColor` or `RGBBackColor` function to set a color as the foreground or background color, Color QuickDraw calls on the Color Manager to find the color in the CLUT of the current graphics device that maps most closely to the color you submitted. The Color Manager returns the index to the best color, and Color QuickDraw puts the value in the `fgColor` field of your application's `CGrafPort` data structure.

Note

QuickDraw and the Palette Manager also use Color Manager functions to change the entries in a device's CLUT. Applications that do color painting and animation need to control the precise colors they use; they should use the Palette Manager to allocate colors. Palette Manager functions operate transparently across multiple screens, but Color Manager functions do not. ♦

The sections that follow describe how the Color Manager selects from colors available on the CLUT to satisfy color requests.

Graphics Devices

The Color Manager, like Color QuickDraw, accesses a particular graphics device through a data structure known as a `GDevice` data structure. Each `GDevice` data structure stores information about a particular graphics device; after this data structure is initialized, the device itself is known to the Color Manager and QuickDraw through that `GDevice` data structure.

A graphics device, represented by a `GDevice` data structure, is a logical device that the software treats identically whether it is a video card, a display device, or an offscreen graphics world. The Color Manager uses fields of the `GDevice` data structure to track application-defined functions and to manage the device CLUT. (See the chapter “Graphics Devices” in *Inside Macintosh: Imaging With QuickDraw* for additional information about the `GDevice` data structure.)

The Color Manager uses three fields of a `GDevice` data structure to track application-defined custom search and complement functions. The `gdSearchProc` field contains a handle to a list of search functions, and the `gdCompProc` field contains a handle to a list of color complement functions. In each list the Color Manager's default function is at the end of the list, to be used if there are no others, or if they fail. The `gdId` field contains an identifier to connect a particular custom function with the application that created it.

Two fields of the `GDevice` data structure contain handles to tables that the Color Manager uses to specify and look up the colors in a device's CLUT. The `gdPMap` field contains a handle to the pixel map for the device. The pixel map in turn contains a handle to the `ColorTable` data structure that contains the colors currently loaded in the CLUT. The `gdITable` field contains a handle to an inverse table that the Color Manager creates and maintains as a means of quickly finding colors in the color table.

Color Tables

The complete set of colors available at a given time for an indexed-pixel device is contained in a `ColorTable` data structure. (The `ColorTable` data structure is described in the chapter “Color QuickDraw” in *Inside Macintosh: Imaging With QuickDraw*.)

```
struct ColorTable {
    long      ctSeed;      /* unique identifier for table */
    short     ctFlags;     /* high bit is set for a */
                        /* GDevice, clear for a PixMap */
    short     ctSize;      /* number of entries in table minus 1 */
    CSpecArray ctTable;    /* array[0..0] of ColorSpec records */
};
typedef struct ColorTable ColorTable;
```

The `ctSeed` field contains a version identifier for the color table. Its value is a unique number higher than `minSeed`, a predefined constant with a value of 1023. (If a color table is created from a resource, its resource number becomes the initial `ctSeed`.) Values of 1023 and below are reserved for standard color tables defined by Color QuickDraw. Color tables that are part of a `GDevice` data structure always have the high bit of the `ctFlags` field set. (Color tables that are part of pixel maps not associated with a `GDevice` data structure have this bit clear.)

The `ctTable` field contains an array of `ColorSpec` entries. The type `ColorSpec` consists of an integer value and a color, as shown in the following specification.

```
struct ColorSpec {
    short     value;      /* color representation */
    RGBColor  rgb;        /* color value */
};
typedef struct ColorSpec ColorSpec;
```

In color tables for screen devices, the Color Manager and Palette Manager use the `value` field to contain color matching and protection information; in such tables, the index for a particular color is determined by its position in the table, *not* by the contents of the `value` field.

Inverse Tables

The Color Manager constructs **inverse tables** in order to remap the information in a device's color lookup table so that, when Color QuickDraw supplies a color, the Color Manager can quickly return the index to the closest color available in the CLUT.

Note

The material in this section is provided for developers who are planning to create their own color-mapping methods; few programs need to use inverse tables directly, and even fewer need to create their own. ♦

When an application sets the foreground color with the `RGBForeColor` function, and then draws using that color (with a `LineTo` function, for example), Color QuickDraw must determine how to deal with the 48 bits of that `RGBColor` data structure. If the line is drawn into a picture, Color QuickDraw can store all 48 bits of color information along with the line-drawing codes. But when it is drawn into a pixel map, some color information must be discarded, because even the deepest pixel map can store only 24 bits of color information per pixel.

Drawing into direct pixel maps, in which each pixel value directly specifies the red, green, and blue components of a color, is straightforward: Color QuickDraw truncates low-order bits from each 16-bit color component of the `RGBColor` data structure. Direct pixel maps can be either 16 or 32 bits deep, with each pixel value containing 15 or 24 bits of color information (5 or 8 bits per component, with the other bits unused).

Indexed pixel maps don't contain `RGBColor` data structures; they contain indexes to a color table where the `RGBColor` data structures are stored. The color table of a `GDevice` data structure reflects the current CLUT values for that device. When an application requests a color for an indexed device, Color QuickDraw calls on the Color Manager to determine which color currently in the `GDevice` data structure's color table (and hence the device's CLUT) comes closest to the requested color. The Color Manager function `Color2Index` returns the index for a given color, and Color QuickDraw stores that index in the pixel map. Indexed pixel maps can be 1, 2, 4, or 8 bits deep. (A 1-bit pixel map is effectively the same as a bitmap.) The most common device CLUT has 256 entries, each of which can be addressed by a pixel map that is 8 bits deep.

Color Manager

Determining the best color choice out of 256 candidates can take a lot of processing time. To speed up processing, the Color Manager builds an inverse table for every indexed device that the Slot Manager finds at startup. The Color Manager stores a handle to the inverse table in the `gdITable` field of the `GDevice` data structure. An inverse table organizes the information in a device's CLUT so that, given an `RGBColor` data structure, the index to the best color can be found quickly. Its format is the inverse of a color table: instead of a collection of `RGBColor` data structures that can be looked up by an index, the inverse table contains a collection of color table indexes that can be looked up by an RGB color value.

The format of an inverse table can be illustrated compactly with a hypothetical color world in which red, green, and blue components are only 1 bit deep each. In Figure 7-1, the eight possible RGB color values of such a world are the indexes into a table whose *entries* consist of indexes to the best colors in the CLUT for that RGB color value.

Figure 7-1 Sample inverse table

R G B	Inverse table entry
0 0 0	Index for best black
0 0 1	Index for best blue
0 1 0	Index for best green
0 1 1	Index for best cyan
1 0 0	Index for best red
1 0 1	Index for best magenta
1 1 0	Index for best yellow
1 1 1	Index for best white

Color Manager

In Color QuickDraw's world, inverse tables may use 3, 4, or 5 bits for each color component. The number of bits used is called the *resolution* of the inverse table. The size of the table depends on its resolution; the table must be large enough to accommodate every color combination at a given resolution. For example, the number of combinations in a table of resolution 3 is 512, so an inverse table of resolution 3 has 512 entries, of one byte each. (Here's how to calculate the 512: there are 3 bits each for red, green, and blue, totaling 9 bits, the permutations of which are 2^9 or 512.) Tables of resolution 4 and 5 occupy approximately 4 KB and 32 KB, respectively.

The format of an inverse table follows.

```
struct ITab {
    long          iTabSeed;          /* copy of color table seed */
    short         iTabRes;           /* resolution of table: 3, 4, or 5 */
    unsigned char iTTable[1];       /* byte-length color */
                                   /* table index values */
};
typedef struct ITab ITab;
```

▲ **WARNING**

Because the format of inverse tables is subject to change in the future, or may not be present on certain devices, applications should not assume the structure of the inverse table's data. ▲

The first entry of the `iTabTable` array, at location 0 (red, green, and blue values of 0000, 0000, 0000 in a table of resolution 4) contains the index to black or the nearest color to it in the CLUT. Similarly, the last entry in a table of resolution 4, at location 4095 (1111, 1111, 1111), contains the CLUT index to white.

Since the largest CLUT has only 256 entries, many inverse table entries contain duplicates of other entries. For example, in an inverse table of resolution 4, the first several entries may all point to the index for black, the last few entries may all point to the index for white, and even various entries in the middle may point to white or black, as illustrated in Figure 7-2.

Figure 7-2 An inverse table of resolution 4

R	G	B	Inverse table entry
0000	0000	0000	Index for best black
0000	0000	0001	Index for best black
0000	0000	0010	Index for best black
0000	0000	0011	Index for best dark blue
0000	0000	0100	Index for best dark blue
0000	0001	0000	Index for best black
0000	0001	0001	Index for best black
1110	1111	1110	Index for best white
1110	1111	1111	Index for best white
1111	1111	1100	Index for best light yellow
1111	1111	1101	Index for best white
1111	1111	1110	Index for best white
1111	1111	1111	Index for best white

Note
Finding indexes by means of inverse tables is the Color Manager’s default method for color mapping. Applications with special color processing requirements may need to override the code for inverse table mapping with custom functions that have special mapping rules. ♦

Inverse Tables in Action

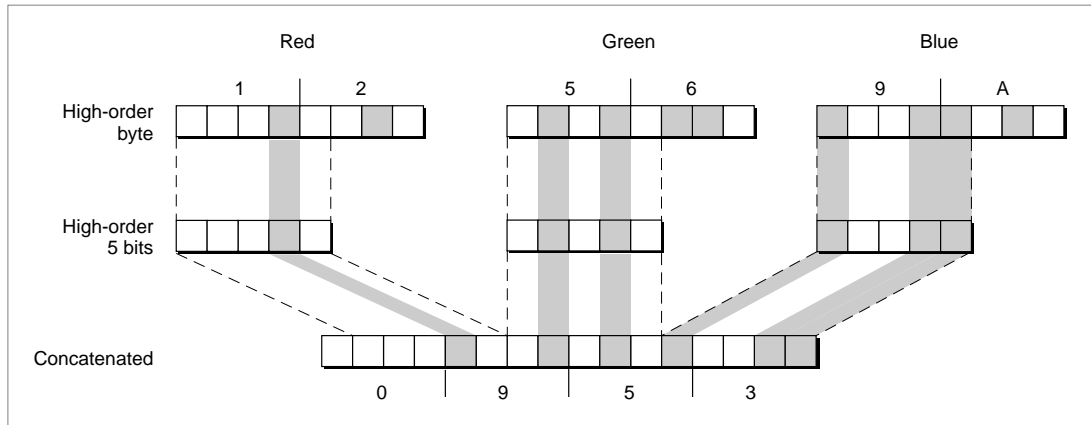
When Color QuickDraw supplies a color, the Color Manager truncates the red, green, and blue values to the size of the table resolution, and concatenates the results to form an index into the table. At that location, the Color Manager, when it built the inverse table, stored the index to the best-mapping color in the CLUT.

Table 7-1 shows two indexes into two inverse tables, one of resolution 4, the other of resolution 5, when the `Color2Index` function is given a color of 0x123456789ABC:

Table 7-1 Sample inverse table indexes

Table resolution	RGBColor data structure	Inverse table index	Table size
4-bit	red=0x1234, green=0x5678, blue=0x9ABC	0x0159	$2^{12} = 4 \text{ KB}$
5-bit	red=0x1234, green=0x5678, blue=0x9ABC	0x0953	$2^{15} = 32 \text{ KB}$

The Color Manager derived the 4-bit index by taking the top 4 bits of each color component. The 5-bit index is more subtle. Figure 7-3 illustrates the steps of truncation and concatenation that result in the value 0x0953.

Figure 7-3 Creating an inverse table index

The Color Manager truncates the high-order 8 bits of each color component to the high-order 5 bits, and then concatenates those 15 bits. The resulting value, 0x0953 in hexadecimal, is the index into the inverse table.

Hidden Colors

Colors that are close in the red, green, and blue color space can become hidden when they differ only in low-order bits that weren't used to construct the inverse table index. For example, even if the CLUT were loaded with 256 levels of gray, a 4-bit inverse table can only discriminate among 16 of the levels. (In the 4096 permutations there are only 16 places where the red, green, and blue values are equal.)

To solve this problem, inverse tables carry additional information about how to find colors that differ only in the less significant bits. (The information is kept private to the Color Manager.) As a result, when the `Color2Index` function is called, it can find the best color to the full 48-bit resolution available in a `ColorSpec` data structure. Since examining the extra information takes time, certain parts of Color QuickDraw, notably the arithmetic transfer modes, don't use this information and hence won't find the hidden colors.

Building Inverse Tables

The Color Manager builds an inverse table for each graphics device at startup. The data in inverse tables remains valid as long as the CLUT from which it was built remains unchanged.

The Color Manager rebuilds a table whenever Color QuickDraw, the Color Picker Manager, or the Palette Manager requests colors from a graphics device whose CLUT has changed. To determine whether an inverse table has been changed, the Color Manager compares the `ctSeed` field of the current `GDevice` data structure's color table against the `iTabSeed` field of that graphics device's inverse table. Each function that modifies the `ColorTable` data structure (with the exception of the `RestoreEntries` function) increments the value in its `ctSeed` field. If `ctSeed` and `iTabSeed` don't match, the Color Manager invalidates the inverse table and rebuilds it.

Note

Under normal circumstances, your application does not know when the Color Manager has invalidated an inverse table. The method of invalidation is the same as the one that Color QuickDraw uses to invalidate expanded patterns and cursors. ♦

If your application modifies a color table entry directly (which is not recommended), it should call the `CTabChanged` function, described in the chapter "Color QuickDraw" in *Inside Macintosh: Imaging With QuickDraw*. `CTabChanged` changes the `iTabSeed` value so that the Color Manager rebuilds the table the next time a pixel index is requested.

The Color Manager builds or rebuilds an inverse table in the following manner:

1. It builds a table of all possible RGB color values taken 3, 4, or 5 bits at a time.
2. For each position in the table, it attempts to find the closest match.
3. It adds information to get a better resolution.

The Color Manager builds an inverse table with the `MakeITable` function, which supports only 3-bit, 4-bit, and 5-bit resolution. (Five bits is the maximum resolution, because the indexes into a 6-bit table would have to be 18 bits long, more than a full word.) In most cases, when your application sets colors in a graphics port using the `RGBForeColor` and `RGBBackColor` functions or uses `CopyBits` to transfer pixel maps, inverse tables of 4 bits, the default, are sufficient. If you use arithmetic transfer modes with color tables that have

closely spaced colors, the screen appearance may be improved by specifying inverse tables at 5-bit resolution, which may uncover some hidden colors.

Using the Color Manager

Color QuickDraw uses the Color Manager for several tasks: to determine the index value of the best mapping color available in a device CLUT, to determine whether a specific color exists in a CLUT, to change the color in a CLUT entry, and to find the complement of a color. Color QuickDraw's use of the Color Manager is transparent to most applications.

The vast majority of applications don't need to use Color Manager functions directly. Palette Manager functions can manipulate the colors of all the CLUTs attached to a system, and the Palette Manager takes care to minimize disturbances to other windows when your application needs to change the values in one or more CLUTs.

Some applications may need to modify the algorithms by which the Color Manager determines the best mapping color in a CLUT or inverts a color. A few applications that are designed to run in specialized environments, where the machine is dedicated to a single task, for example, may need to directly modify a device CLUT.

Customizing Search Functions

Specialized color applications may need to modify the way in which the Color Manager determines the best-mapping color in a CLUT. A custom search function you provide can implement its own mapping rules. For instance, a color-selection application might want to map all levels of green to a single green on a monitor.

To do this, you write your own color search function and use the `AddSearch` function to install it in the list of search functions that the Color Manager maintains. Then when your application or Color QuickDraw calls the `Color2Index` function, the Color Manager calls on your search function to find and return the index for the best-mapping color. Your function should be defined as:

```
Boolean MySearchProc (RGBColor *rgb, long *position);
```

Color Manager

If your search function decides to act on the color, it returns the index of the desired color in the `position` parameter, and returns `true` for the function value. Otherwise, it should return `false` as the function value, and pass the input color back to the Color Manager in the `rgb` parameter. The Color Manager then looks to the next search function in the list, and if there are no other custom functions it uses the normal inverse table mechanism.

The functions are chain elements in a linked list beginning in the `gdSearchProc` field of the current `GDevice` data structure. Each link is an `SProcRec` data structure, which has the format shown below.

```
struct SProcRec {
    Handle          nxtSrch;    /* handle to next sProcRec */
    ColorSearchProcPtr srchProc; /* pointer to search function */
};
typedef struct SProcRec SProcRec;
```

The Color Manager provides functions to add and delete custom functions from the linked list. You can install any number of search functions in the list. The Color Manager gives each function the chance to act or pass on the color until one returns `true`. Since each device is a shared resource, you can use the `SetClientID` function to set the `gdID` field of the `GDevice` data structure to identify your application as the caller to your search functions. When your application is finished mapping colors, it should remove your custom search function, by calling the `DelSearch` function.

For example, Listing 7-1 shows a function that adds a custom search function before drawing with a user-supplied color.

Listing 7-1 Adding and using a custom search function

```
void MyDrawUserColor(void)
{
    ColorSearchProcPtr MyColorSearch;
    RGBColor           *UserColor;
    Rect               *sampleRect;

    AddSearch(MyColorSearch);
    RGBForeColor(UserColor);
}
```

Color Manager

```

    FrameRect(sampleRect);
    DelSearch(MyColorSearch);
}

```

The `MyColorSearch` function could entirely replace the Color Manager's search algorithm by accepting the `RGBColor` data structure, doing a search, and then returning the index value and `true`. It could also merely modify the color and then let the Color Manager perform its inverse table lookup. For example, the `MyColorSearch` function could accept the `RGBColor`, do an arithmetic add with a gray color to lighten all three components, and then return the `RGBColor` data structure in the `rgb` VAR parameter and return `false` as the function value. The Color Manager then uses the returned color as the input for its default search.

Customizing Complement Functions

Some specialized color applications may need to modify the way in which the Color Manager calculates the inverse, or complement, of a color. The Color Manager's default method for inverting a color is to invert the bit values of the `RGBColor` data structure. For colors in which at least one component has very low or very high values this method produces expected (or at least discernible) results. A pure green, for example, with RGB color values of 0x0000, 0xFFFF, and 0x0000, inverts to magenta, with RGB color values of 0xFFFF, 0x0000, and 0xFFFF. But midrange values do not change by much: a component value of 0x7FFF, for example, inverts to 0x8000, which is an indiscernible difference. A gray value always inverts to another gray, which may or may not be what your application needs.

Your application can supply a custom complement function to find the inverse of a specified color. The Color Manager keeps complement functions in a linked list beginning in the `GDevice` data structure's `gdCompProc` field. Each link is a `CProcRec` data structure, which has the format shown below.

```

struct CProcRec {
    Handle                nxtComp;    /* handle to next CProcRec */
    ColorComplementProcPtr compProc;  /* pointer to complement
                                     function */
};
typedef struct CProcRec CProcRec;

```

You install a complement function with the `AddComp` function. When your application is done using it, be sure to remove it with the `DelComp` function.

Managing the Device CLUT

The Color Manager functions that directly modify a device CLUT should be used with care, because in a multitasking environment, changes to a device CLUT affect colors used by the system software and other application windows.

Note

The Color Manager functions described next are designed to operate on a single `GDevice` data structure. The Palette Manager can perform most of these operations across multiple `GDevice` data structures. Since the Palette Manager provides more general and portable functionality, applications should use Palette Manager functions whenever possible. ♦

Use the `SetEntries` function to change any of the entries in a device's CLUT. `SetEntries` changes the `CTSeed` field value, so the Color Manager knows to rebuild the inverse table.

The `SaveEntries` and `RestoreEntries` functions can make temporary changes to the color table under very specialized circumstances (for example, from a color selection dialog box within an application). Applications don't need these functions under normal circumstances. For example, if you use the Palette Manager functions `GetEntryColor` and `SetEntryColor` on a palette, rather than directly changing a device's color table, the Palette Manager ensures that colors used by other applications and the system are not affected.

You can use the `SaveEntries` function to copy any combination of `ColorSpec` data structures into a color table. `RestoreEntries` replaces the table created by `SaveEntries` in the graphics device. Unlike `SetEntries`, these functions don't perform invalidations of the device's color table, so they avoid causing invalidations of cached data structures. If you use these functions, your application must take responsibility for rebuilding and restoring auxiliary structures as necessary.

By convention, when using `SetEntries` or `RestoreEntries`, you should store white at color table position 0, and black in the last color table position available, whether it is 1, 3, 15, or 255. The Palette Manager also enforces this convention. The most commonly found CLUT contains 256 entries, but a

Color Manager

4-entry CLUT can illustrate setting entry values more concisely. Here's a `CSpecArray` data structure called `MyColors` containing four RGB color values:

Table 7-2 A sample `CSpecArray` data structure

Value	Red	Green	Blue
0	FFFF	FFFF	FFFF
1	FFFF	0000	0000
2	8888	8888	8888
3	0000	0000	0000

Then you could use the `SetEntries` function to fill the four entries of the CLUT starting at 0 with white at the first entry, a bright red at the second, a gray at the third, and black at the last entry:

```
SetEntries(0, 4, MyColors);
```

Color Manager functions maintain special information in device color tables. Using the `ProtectEntry` and `ReserveEntry` functions, an entry may be protected, which prevents `SetEntries` from further changing the entry, or reserved, which makes the entry unavailable to be mapped by the `RGBForeColor` and `RGBBackColor` functions. Functions that change the device table (`SetEntries`, `ProtectEntry`, and `ReserveEntry`, but not `RestoreEntries`) perform the appropriate invalidations of QuickDraw data structures. Your application must then redraw where necessary.

Use the `RealColor` function to determine whether a particular color exists in a color table. `RealColor` determines whether any color in the CLUT for the current `GDevice` data structure matches the given color to the resolution of the device's inverse table. For example, if the device's inverse table is set to a resolution of 4 (the default), `RealColor` returns `true` if any color in the CLUT exactly matches the top four bits of each component. You could make `RealColor` match to the top 5 bits by rebuilding the inverse table: call `MakeITable` with a `res` parameter value of 5.

Color Manager

The `Color2Index` function returns the index in the current device's CLUT that is the best match to the requested color. The `Index2Color` function performs the opposite function—it returns the color of a particular index value. These functions can be useful when making copies of the screen frame buffer. Use the `GetSubTable` function to obtain a set of indexes for a CLUT; it calls the `Color2Index` function for each input color.

Summary of the Color Manager

Constants and Data Types

CONST

```

struct MatchRec {
    unsigned short  red;
    unsigned short  green;
    unsigned short  blue;
    long            matchData;
};
typedef struct MatchRec MatchRec;

struct ITab {
    long            iTabSeed;        /* copy of CTSeed from source CTable */
    short           iTabRes;         /* bits/channel resolution of iTable */
    unsigned char   iTTable[1];     /* byte colortable index values */
};
typedef struct ITab ITab;
typedef ITab *ITabPtr, **ITabHandle;

struct SProcRec {
    Handle          nxtSrch;         /* handle to next SProcRec */
    ColorSearchProcPtr srchProc;     /* pointer to search function */
};
typedef struct SProcRec SProcRec;
typedef SProcRec *SProcPtr, **SProcHndl;

struct CProcRec {
    Handle          nxtComp;         /* handle to next CProcRec */
    ColorComplementProcPtr compProc; /* pointer to complement function */
};
typedef struct CProcRec CProcRec;
typedef CProcRec *CProcPtr, **CProcHndl;

```

Color Manager

```

struct ReqListRec {
    short reqLSize;          /*request list size*/
    short reqLData[1]       /*request list data*/
};
typedef struct ReqListRec ReqListRec;

```

Color Manager Functions

Managing Colors

```

pascal long Color2Index      (const RGBColor *myColor);
pascal void Index2Color     (long index,
                             RGBColor *aColor);

pascal void InvertColor      (RGBColor *myColor);
pascal Boolean RealColor     (const RGBColor *color);
pascal void GetSubTable      (CTabHandle myColors,
                             short iTabRes,
                             CTabHandle targetTbl);

pascal void MakeITable       (CTabHandle cTabH,
                             ITabHandle iTabH,
                             short res);

```

Managing Color Tables

```

pascal long GetCTSeed        (void);
pascal void ProtectEntry     (short index,
                             Boolean protect);

pascal void ReserveEntry     (short index,
                             Boolean reserve);

pascal void SetEntries       (short start,
                             short count,
                             CSpecArray aTable);

pascal void SaveEntries      (CTabHandle srcTable,
                             CTabHandle resultTable,
                             ReqListRec*selection);

```

```
pascal void RestoreEntries      (CTabHandle srcTable,  
                                CTabHandle dstTable,  
                                ReqListRec *selection);
```

Operations on Search and Complement Functions

```
pascal void AddSearch           (ColorSearchProcPtr searchProc);  
pascal void AddComp             (ColorComplementProcPtr compProc);  
pascal void DelSearch           (ColorSearchProcPtr searchProc);  
pascal void DelComp             (ColorComplementProcPtr compProc);  
pascal void SetClientID         (short id);
```

Application-Defined Functions

```
pascal Boolean MySearchProc     (RGBColor *rgb,  
                                long *position);  
pascal void MyCompProc          (RGBColor *rgb);
```


ColorSync Manager Backward Compatibility

The ColorSync Manager version 2.0 replaces the earlier version of the product called ColorSync 1.0. This appendix describes backward compatibility support for ColorSync 1.0 functions, profiles, and CMMs provided by the ColorSync Manager version 2.0. Hereafter, the name ColorSync Manager is used to mean ColorSync Manager version 2.0.

The ColorSync Manager provides backward compatibility with ColorSync 1.0 by supporting the ColorSync 1.0 API and ColorSync 1.0 profiles.

You should use the ColorSync 1.0 API with ColorSync 1.0 profiles and the ColorSync Manager API with version 2.0 profiles. However, there are exceptional cases for which you must use a ColorSync 1.0 profile with the ColorSync Manager.

In addition to describing ColorSync Manager backward compatibility with ColorSync 1.0, this appendix explains how to use the ColorSync Manager API for color matching between a ColorSync 1.0 profile and a version 2.0 profile.

ColorSync 1.0 API Support

For its first release, the ColorSync Manager will continue to implement fully the ColorSync 1.0 API, including the ColorSync 1.0 profile responder. Existing applications and drivers written to the ColorSync 1.0 API, and ColorSync 1.0 profiles, CMMs, and QuickDraw GX 1.0 will continue to work properly with the first release of the ColorSync Manager.

Although the ColorSync Manager continues to support use of the profile responder for ColorSync 1.0, this feature is not supported by the ColorSync Manager API.

ColorSync 1.0 Profile Support

For the first release of the product, the ColorSync Manager will continue to support the use of ColorSync 1.0 profiles. Apple Computer strongly recommends that you use the ColorSync 1.0 API with ColorSync 1.0 profiles, if possible. For example, always use the ColorSync 1.0 API to match colors between the color gamuts of two devices if both devices have ColorSync 1.0 profiles.

However, there are times when you may need to use a ColorSync 1.0 profile with the ColorSync Manager API. The ColorSync Manager's robust backward-compatibility support allows you to do this. For example, a document containing an image to be color matched may include an embedded ColorSync 1.0 source profile for the image. To match the colors of the source image to a device that has a version 2.0 profile, you must use the ColorSync Manager API because the ColorSync 1.0 API cannot gain access to a version 2.0 profile.

One of the main differences between ColorSync 1.0 and the ColorSync Manager is the profile format used. The ColorSync Manager accommodates ColorSync 1.0 profiles so that you can use them with it when you must. Before describing how to use a ColorSync 1.0 profile with the ColorSync Manager, this section explains the differences between the ColorSync 1.0 profile format and the version 2.0 profile format defined by the International Color Consortium (ICC) and used by the ColorSync Manager.

ColorSync 1.0 Profiles and Version 2.0 Profiles

The ColorSync 1.0 profile format was designed by Apple Computer. This profile is memory resident and follows an internal structure based on tables. Although it is an open format, it is not an industry standard.

ColorSync Manager Backward Compatibility

The ICC profile format implemented in the ColorSync Manager is significantly different from the profile format implemented for ColorSync 1.0. The version 2.0 profile format is specified by the ICC and provides an industry standard that allows for interoperability across platforms and devices. A version 2.0 profile created for a particular device can be used on systems running different operating systems.

Because the ColorSync 1.0 and version 2.0 profile formats differ, the ColorSync Manager must resolve any compatibility issues involving accessing profiles and color matching between profiles.

How ColorSync 1.0 Profiles and Version 2.0 Profiles Differ

A ColorSync 1.0 profile is smaller than a version 2.0 profile, which allows for it to reside in memory. It is a handle-based profile. A version 2.0 profile as implemented by the ColorSync Manager is commonly file based, but it can also be memory based. You use an abstract internal data structure, called a profile reference, to access a version 2.0 profile.

A ColorSync 1.0 profile contains a header, a copy of the Apple `CMProfileChromaticities` record, profile response data for the associated device, and a profile name string for use in dialog boxes. Custom profiles may also have additional, private data. ColorSync 1.0 defines the following profile data structure:

```
struct CMProfile, *CMProfilePtr, **CMProfileHandle {
    CMHeader                header;
    CMProfileChromaticities  profile;
    CMProfileResponse        response;
    IString                  profileName;    /* variable length */
    char                    customData[anyNumber];
                                /* optional custom CMM data */
};
```

The response data fields contain nine tables. The first table is for grayscale values. The next three are red, green, and blue values, followed by three for cyan, magenta, and yellow values. The eighth and ninth tables are for CMYK printers requiring undercolor removal and black generation data.

ColorSync Manager Backward Compatibility

The ColorSync 1.0 profile header and the version 2.0 profile header contain many fields in common. However, the ColorSync 1.0 profile header contains fields that reflect its table-based structure. ColorSync 1.0 defines the following profile header data structure:

```
struct CMHeader{
    unsigned long    size;          /* this is the total size of the
                                   profile including custom data */
    OSType           CMMType;       /* preferred CMM */
    unsigned long    applProfileVersion; /* profile version */
    OSType           dataType;
    OSType           deviceType;
    OSType           deviceManufacturer;
    unsigned long    deviceModel;
    unsigned long    deviceAttributes[2];
    unsigned long    profileNameOffset; /* offset to profile name
                                         from top of data */
    unsigned long    customDataOffset; /* offset to custom data
                                         from top of data */

    CMMatchFlag      flags;
    CMMatchOption     options;
    XYZColor          white;
    XYZColor          black;
};
```

As implemented in the ColorSync Manager, a version 2.0 profile is a tagged-element structure that begins with a profile header. A version 2.0 profile supports use of lookup table transforms that allow for faster processing. The ColorSync Manager defines the following profile header data structure for version 2.0 profiles:

```
struct CM2Header {
    unsigned long    size;
    OSType           CMMType;
    unsigned long    profileVersion;
    OSType           profileClass;
    OSType           dataColorSpace;
    OSType           profileConnectionSpace;
    CMDateTime       dateTime;
    OSType           CS2profileSignature;
    OSType           platform;
```

ColorSync Manager Backward Compatibility

```

        unsigned long      flags;
        OSType             deviceManufacturer;
        unsigned long      deviceModel;
        unsigned long      deviceAttributes[2];
        unsigned long      renderingIntent;
        CMFixedXYZColor     white;
        char                reserved[48];
};

```

CMMs and Mixed Profiles

Although the ColorSync Manager API supports using a mix of ColorSync 1.0 and version 2.0 profiles, the success of a matching session involving a ColorSync 1.0 profile depends on the CMM component performing the process. Third-party CMMs may choose not to support ColorSync 1.0 profiles. The Apple-supplied default CMM is able to establish a matching session involving one or more ColorSync 1.0 profiles.

For device-linked profiles, you must include only version 2.0 profiles. You cannot mix ColorSync 1.0 and version 2.0 profiles in a device-linked profile.

Using the ColorSync Manager API With ColorSync 1.0 Profiles

Despite differences between the version 2.0 and ColorSync 1.0 profile formats, you can use most of the ColorSync Manager 2.0 functions to gain access to ColorSync 1.0 profiles and their contents and to color match to and from the two disparate profile formats, if necessary. The ColorSync Manager makes this possible.

You can open a reference to a ColorSync 1.0 profile using the ColorSync Manager functions and special data structures that accommodate both profile styles. You can also match the colors of an image expressed in the color gamut of one device whose characteristics are described by a ColorSync 1.0 profile to the colors within the gamut of another device whose characteristics are described by a version 2.0 profile.

IMPORTANT

If you are color matching between devices that both use ColorSync 1.0 profiles, you should use the ColorSync 1.0 API for the process. ▲

This section describes

- which ColorSync Manager functions you cannot use for ColorSync 1.0 profiles
- how you can use the ColorSync Manager with ColorSync 1.0 profiles

ColorSync Manager Functions Not Supported for ColorSync 1.0 Profiles

You cannot use the ColorSync Manager's `CMUpdateProfile` function to update a ColorSync 1.0 profile. The ColorSync Manager does not provide functions for profile version conversions. This is the domain of profile-building tools and calibration applications.

The ColorSync Manager API includes a set of functions used to search the ColorSync™ Profiles folder for specific profiles that meet search criteria. These functions act on version 2.0 profiles only. If the ColorSync™ Profiles folder contains ColorSync 1.0 profiles, these functions do not acknowledge them or return results that include them. The ColorSync Manager search functions, which are not supported for ColorSync 1.0 functions, are the `CMNewProfileSearch`, `CMUpdateProfileSearch`, `CMDisposeProfileSearch`, `CMSearchGetIndProfile`, and `CMSearchGetIndProfileFileSpec`.

You cannot use the ColorSync Manager's `NCMUseProfileComment` function to automatically generate the picture comments required to embed a ColorSync 1.0 profile. This function is designed to work with version 2.0 profiles only.

Using ColorSync 1.0 Profiles With the ColorSync Manager

To match an image contained in a document that includes the image's embedded ColorSync 1.0 source profile to the color gamut of a printer defined by a version 2.0 profile, you must use the ColorSync Manager. The ColorSync Manager is equipped to contend with both profile formats.

The subsections that follow explain how to obtain a reference to the ColorSync 1.0 profile, get the profile's header, and get its synthesized tags.

Opening a ColorSync 1.0 Profile

To use a ColorSync 1.0 profile with the ColorSync Manager API, you must obtain a reference to the profile. Obtaining a reference to the profile is synonymous with opening the profile for your program's use. If the profile is embedded in a document, you must extract the profile before you can open it.

You can use the ColorSync Manager `CMOpenProfileFile` function to obtain a reference to a ColorSync 1.0 profile. Other ColorSync Manager functions that you use to gain access to the profile's contents or perform color matching based on the profile require the profile reference as a parameter.

Obtaining a ColorSync 1.0 Profile Header

After you obtain a reference to a profile, you can gain access to the profile's contents. To gain access to the contents of any of the fields of a profile header, you must get the entire header. The ColorSync Manager allows you to do this using the `CMGetProfileHeader` function. You pass this function the profile reference and a data structure to hold the returned header. The ColorSync Manager defines the following union of type `CMAAppleProfileHeader`, containing variants for ColorSync 1.0 and version 2.0 ColorSync profile headers for this purpose:

```
union CMAAppleProfileHeader {
    CMHeader          cm1;
    CM2Header         cm2;
};
```

You use the `cm1` variant for a ColorSync 1.0 profile header. You can easily test for the version of a profile header to determine which variant to use because the offset of the header version is at the same place for both ColorSync 1.0 profiles and version 2.0 profiles.

Obtaining ColorSync 1.0 Profile Elements

The ColorSync Manager provides four tags to allow you to obtain four ColorSync 1.0 profile elements pointed to from the profile header or contained outside the header. To obtain the profile element, you specify its associated tag signature as a parameter to the `CMGetProfileElement` function along with the profile reference. The ColorSync Manager provides the following enumeration that defines these tags:

```
enum {
    cmCS1ChromTag    = 'chrn',
    cmCS1TRCTag      = 'trc ',
    cmCS1NameTag      = 'name',
    cmCS1CustTag      = 'cust'
};
```

'chrn'	Profile chromaticities tag signature. Element data for this tag specifies the XYZ chromaticities for the six primary and secondary colors (red, green, blue, cyan, magenta, and yellow).
'trc '	Profile response data tag signature. Element data for this tag specifies the profile response data for the associated device.
'name'	Profile name string tag signature. Element data for this tag specifies the profile name string. This is an international string consisting of a Macintosh script code followed by a length byte and up to 63 additional bytes composing a text string that identifies the profile.
'cust'	Custom tag signature. Element data for this tag specifies the private data for a custom CMM.

Embedding ColorSync 1.0 Profiles

In ColorSync 1.0, picture comment types `cmBeginProfile` and `cmEndProfile` are used to begin and end a picture comment.

The `cmEnableMatching` and `cmDisableMatching` picture comments are used to begin and end color matching in ColorSync 1.0 and the ColorSync Manager.

ColorSync 1.0 Functions With Parallel ColorSync Manager Counterparts

Four of the functions supported by the ColorSync 1.0 API have been implemented in the ColorSync Manager API. The ColorSync Manager version of these functions follows the ColorSync Manager API style. For example, a parameter used to specify a profile takes a profile reference.

It is easy to spot a ColorSync Manager function that is a new version of a ColorSync 1.0 function, because the function's name begins with an uppercase letter *N*, signifying that it is new.

If you are writing a new ColorSync-supportive program, you should always use the new ColorSync Manager functions. The ColorSync 1.0 version of these functions will continue to be supported for only the first release of the ColorSync Manager.

Here are the four ColorSync 1.0 functions and their ColorSync Manager counterpart functions:

Table A-1 ColorSync 1.0 functions and their ColorSync Manager counterparts

ColorSync 1.0 function	ColorSync Manager function
pascal CWNewColorWorld (CMWorldRef *cw, CMProfileHandle src, CMProfileHandle dst);	pascal CLError NCWNewColorWorld (CMWorldRef *cw, CMProfileRef src, CMProfileRef dst);
pascal CLError CMBeginMatching (CMProfileHandle src, CMProfileHandle dst, CMMatchRef *myRef);	pascal CLError NCMBeginMatching (CMProfileRef src, CMProfileRef dst, CMMatchRef *myRef);
pascal void CMDrawMatchedPicture (PicHandle myPicture, CMProfileHandle dst, Rect *myRect);	pascal void NCMDrawMatchedPicture (PicHandle myPicture, CMProfileRef dst, Rect *myRect);
pascal CLError CMUseProfileComment (CMProfileHandle profile);	pascal CLError NCMUseProfileComment (CMProfileRef prof, unsigned long flags);

Glossary

absolute colorimetric matching A **rendering intent** that is used for a device-independent color space in which the result is an idealized print viewed on a perfect paper having a large dynamic range and color gamut. In reality, paper cannot reproduce densities less than a particular minimum density.

abstract profile A profile that allows applications to perform special color effects independent of the devices on which the effects are rendered.

additive color theory The process of mixing red, green, and blue lights, which are each approximately one-third of the visible spectrum. Additive color theory explains how red, green, and blue light can be added to make white light.

animated color A color that the Palette Manager uses for special animation effects. Animated colors work only on devices that have a color table; that is, they do not work on direct devices.

application-owned dialog box A dialog box, created by an application, for presenting a color picker.

brightness A term in color theory used to describe differences in the intensity of light reflected from or transmitted by a color image. The hue of an object may be blue, but the adjectives dark or light distinguish the brightness of one object from another. Compare with **hue** and **saturation**.

CIE-based color spaces Color spaces that allow color to be expressed in a device-independent way, unlike RGB colors, which vary with display, and scanner characteristics and CMYK colors, which vary with printer, ink, and paper characteristics. CIE-based color spaces result from work carried out in 1931 by the Commission Internationale d'Eclairage (CIE). These color spaces are also referred to as device-independent color spaces.

CMM See **color management module**.

color channel See **color component**.

color component A dimension of a color value expressed as a numeric value. For the ColorSync Manager, depending on the color space, a color value may consist of one, two, three, four, or eight components, also referred to as channels.

color-component value A value that represents the color of a component. Each component of a color space has a color-component value. A color-component value can vary from 0 to 65,535 (0xFFFF), although the numerical interpretation of that range is different for different color spaces. In most cases, color-component intensities are interpreted numerically as varying between 0 and 1.0. See also **color space** and **color value**.

color conversion The process of converting colors from one color space to another.

color gamut See **gamut**.

color management module A component, also referred to as a CMM, that carries out the actual color matching and gamut-checking processes based on requests resulting from calls a program makes to the ColorSync Manager API. An application or driver can supply its own CMM or it can use the robust default CMM that Apple supplies.

Color Manager A set of system software functions that supply color-selection support for Color QuickDraw. Most applications never need to call the Color Manager directly.

color matching The process of adjusting or *matching* converted colors appropriately to achieve maximum similarity from the gamut of one color space to the other. Color matching always involves color conversion, whereas color conversion may not entail color matching.

color picker Code, implemented as a component, that allows users to select a color from a range of possible colors.

Color Picker Manager A set of system software functions that provide applications with a standard user interface for soliciting color choices from users.

color picker–owned dialog box A dialog box, defined by a color picker, for presenting the color picker.

color space A model for representing color in terms of intensity values; a color space specifies how color information is

represented. It defines a multidimensional space whose dimensions, or components, represent intensity values.

color space profile A profile that contains the data necessary to translate color values, such as CIE into RGB or RGB into CIE, as necessary for color matching. Color space profiles provide a convenient means for CMMs to convert between different nondevice profiles.

ColorSync Manager A set of system software functions that provide applications with device-independent color-matching and color conversion services.

color value A complete specification of a color in a given color space. Depending on the color space used, one, two, three, or four color-component values combine to make a color value.

courteous color A color that accepts whatever value the Color Manager determines is the closest match available in the color table. Compare **tolerant color**.

default system profile The system profile for the display device that the ColorSync Manager includes and uses unless the user selects a different system profile through the ColorSync Manager control panel.

destination profile The profile that describes the characteristics of the output device for which the image is destined. The profile is used to color match the image to the device's gamut.

device-independent color spaces See **CIE-based color spaces**.

device-linked profile A profile that combines multiple profiles, such as various device profiles associated with the creation and editing of an image.

device profile A structure that provides a means of defining the color characteristics of a given device in a particular state.

event forecasters Warnings sent by an application to a color picker about user actions that might adversely affect the color picker.

explicit color A color that specifies an index value in the device's color table rather than an RGB color.

gamut The range of color that a device can produce, also referred to as the device's color gamut.

HLS space A transformation of RGB space that allow colors to be described in terms more natural to an artist. The name *HLS* stands for *hue*, *lightness*, and *saturation*.

HSV space A transformation of RGB space that allow colors to be described in terms more natural to an artist. The name *HSV* stands for *hue*, *saturation*, and *value*.

hue The name of the color that places the color in its correct position in the spectrum. For example, if a color is described as blue, it is distinguished from yellow, red, green, or other colors. Compare with **brightness** and **saturation**.

indexed color space The color space used when drawing with indirectly specified colors.

inhibited color A color that is prevented from appearing on particular screens. Colors can be specifically inhibited on a 2-bit, 4-bit, and 8-bit color or grayscale screen.

interchange color space

Device-independent color spaces that are used for the interchange of color data from the native color space of one device to the native color space of another device.

inverse table A special data structure arranged by the Color Manager in such a manner that, given an arbitrary RGB color, the Color Manager can very rapidly look up its pixel value.

$L^*a^*b^*$ space A nonlinear transformation (that is, a third-order approximation) of the Munsell color-notation system designed to match perceived color difference with quantitative distance in color space.

$L^*u^*v^*$ color space A nonlinear transformation of XYZ space used to create a perceptually linear color space. This color space was designed to match perceived color difference with quantitative distance in color space.

new color In a color picker dialog box, the latest color selected by the user.

original color In a color picker dialog box, the color that the user is about to change.

palette A set of colors optimized for use on display devices with a limited number of colors. A palette defines a set of RGB colors, how they are to be used, and the tolerances within which they must be matched.

perceptual matching A **rendering intent** in which all the colors of a given gamut may be scaled to fit within another gamut. The colors maintain their relative positions, so the relationship between colors is maintained.

pixel value A number used by system software and a graphics device to represent a color. The translation from the color that an application specifies in an `RGBColor` data structure to a pixel value is performed at the time the application draws the color. The process differs for indexed and direct devices.

profile A structure that may contain measurements representing a color gamut, including information such as the lightest and darkest possible tones, and maximum densities for red, green, blue, cyan, magenta, and yellow. The International Color Consortium defines several different types of profiles. Each of these types of profiles must include a different required set of information, but all of these profile types follow the same format.

profile chromaticities Color values that define the extremes of saturation that the device can produce for its primary and secondary colors (red, green, blue, cyan, magenta, yellow).

reference white point A specific definition of what is considered white light represented in terms of XYZ space and usually based on the whitest light that can be generated by a given device.

relative colorimetric matching A **rendering intent** in which the colors that fall within the gamuts of both devices are left unchanged. Relative colorimetric matching allows some colors in both images to be exactly the same, which is useful when colors must match quantitatively. A disadvantage of relative colorimetric matching is that many colors may map to a single color resulting in tone compression.

rendering intent The approach taken when a CMM maps or translates the colors of an image to the color gamut of a destination device. Each profile supports four different rendering intents: **perceptual matching**, **relative colorimetric matching**, **saturation matching**, and **absolute colorimetric matching**.

RGB space A three-dimensional color space whose components are the red, green, and blue intensities that make up a given color.

saturation The degree of hue in a color or a color's strength. A neutral gray is considered to have zero saturation. A saturated red would have the a color similar to apple red. Compare with **brightness** and **hue**.

saturation matching A **rendering intent** in which the relative saturation of colors is maintained from gamut to gamut. Colors outside the gamut are usually converted to colors with the same saturation, but different lightness, at the edge of the gamut.

source profile The profile that is associated with the image and describes the characteristics of the device on which the image was created.

subtractive color theory The process of combining subtractive colorants such as inks or dyes. In this theory colorants of cyan, magenta, and yellow are used to subtract a portion of the white light that is illuminating an object.

system-owned dialog box The default dialog box provided by system software for applications that create custom dialog boxes for color pickers. Applications can make this a box modal, modeless, or moveable modal dialog box.

system profile The profile that defines the color characteristics for the system's display device. The ColorSync Manager provides a control panel to allow the user to specify the system profile for the current display device.

tolerant color A color that accepts—within a specified range—the value that the Color Manager determines is the closest match available in the color table. If there is no match within the specified range, the Palette Manager loads the required color. Compare **courteous color**.

tristimulus values An hypothetical set of primaries, XYZ, set up by the CIE that correspond to the way the eye's retina behaves. The term *tristimulus* comes from the fact that color perception results from the retina of the eye responding to three types of stimuli. After experimentation, the CIE set up a hypothetical set of primaries, XYZ, that correspond to the way the eye's retina behaves.

XYZ color space The fundamental CIE-based color space that allows colors to be expressed as a mixture of the three **tristimulus values** X, Y, and Z.

Yxy color space A color space belonging to the XYZ base family that expresses the XYZ values in terms of x and y chromaticity coordinates, somewhat analogous to the hue and saturation coordinates of HSV space.

Index

A

absolute colorimetric matching 3-23
AddComp function 7-15
additive color 3-6
allocating colors, by Palette Manager 1-17
animated colors 1-12 to 1-13
 allocation of 1-17
 on direct devices 1-13
 in palettes 1-8
 returned to a device 1-13
application-defined functions
 MyBuildAppDialog 2-16
 MyBuildMovableModalSysDialog 2-15
 MyCheckIfPickerCanClose 2-28
 MyColorChangedFunction 2-12
 MyDoMenu 2-25
 MyDoPickerBalloonHelp 2-30
 MyDrawUserColor 7-14
 MyGetDestinationProfile 2-29
 MyPickAColor 2-9
 MyPickerEventFilterFunction 2-11
 MySampleDoEvent 2-22
 MySearchProc 7-13 to 7-15
 MySetDestinationProfile 2-29
ApplicationDialogInfo type 2-16
application-owned dialog box 2-13, 2-16 to 2-17

B

base families for color spaces 3-6
bitmap color-checking request
 defined 5-12
 handling 5-28
bitmap color-matching request, handling 5-27
black generation 3-11
brightness 3-5

C

calibration applications 3-28, 4-58
chromaticity 3-13
CIE-based color spaces 3-11 to 3-14
 defined 3-11
 L*u*v* 3-14
 XYZ 3-12 to 3-13
CLUTs 7-3 to 7-18
CMCloseProfile function 4-23
CMDisposeProfileSearch function 4-51
CMGetScriptProfileDescription
 function 4-20, 4-51
CMGetSystemProfile function 4-20, 4-23
CMMs
 and ColorSync 1.0 profiles A-5
 and device drivers 6-3, 6-8
 defined 3-22, 5-4
 development of 5-3 to 5-41
 for a color world 4-29
 interaction with the Component Manager 5-4
 to 5-6, 5-9 to 5-10
 tasks performed by 5-3, 6-8
CMNewProfileSearch function 4-50
CMOpenProfile function 4-18
CMSearchGetIndProfile function 4-51
CMUnflattenProfile function 4-39
CMY-based color spaces 3-10 to 3-11
 CMY 3-10
 CMYK 3-11
 defined 3-10
CMYK-based color spaces 3-6
CMYK space 3-10 to 3-11
color
 perception of 3-5
 theory, an overview 3-4 to 3-6
Color2Index function 7-6, 7-10, 7-11, 7-13, 7-18
color allocation, by Palette Manager 1-17

- color-changed functions (for color pickers) 2-11
 - to 2-12
- color channels 3-6
- color-checking request
 - defined 5-11
 - handling 5-24
- color complement functions 7-15
- color components 3-6
- color conversion 3-15
- colorimetric matching 3-23
- ColorInfo data type 1-5
- color lookup tables (CLUTs) 7-3 to 7-18
- color management modules. *See* CMMs
- color management systems 3-17
- Color Manager 7-3 to 7-21
 - and Color QuickDraw 7-3
 - and the Color Picker Manager 7-3
 - and the Palette Manager 7-3, 7-16
- color matching 3-15
 - to the display 3-27
- color-matching request
 - defined 5-11
 - handling 5-24
- color picker–defined functions
 - dispatching to 2-34 to 2-37
 - MyColorPickerDispatch 2-34, 2-36
 - MyDoEdit 2-44
 - MyDoEvent 2-41
 - MyDrawPicker 2-41
 - MyGetColor 2-45
 - MyGetDialog 2-40
 - MyGetEditMenuState 2-50
 - MyGetIconData 2-47
 - MyGetItemList 2-40
 - MyGetProfile 2-49
 - MyGetPrompt 2-48
 - MyInitPicker 2-38
 - MyItemHit 2-42
 - MySetColor 2-45
 - MySetProfile 2-49
 - MySetPrompt 2-48
 - MyTestGraphicsWorld 2-39
- Color Picker Manager 2-3 to 2-61
 - and QuickDraw GX 2-3
 - and the Color Manager 7-3
 - and the Color Picker Package 2-3
 - and the ColorSync Manager 2-7
 - and the Component Manager 2-6, 2-31 to 2-40
 - and the Dialog Manager 2-3
 - testing for availability 2-8
- color picker–owned dialog box 2-13, 2-17 to 2-18
- Color Picker Package 2-3, 2-4
- color pickers 2-3 to 2-61
 - and color matching 2-7
 - and help balloons 2-29 to 2-31
 - as components 2-6, 2-31 to 2-40
 - color-changed functions for 2-11 to 2-12
 - color information in custom 2-44 to 2-50
 - creation of 2-31 to 2-33
 - customized dialog boxes for 2-5 to 2-6, 2-13 to 2-29
 - defined 2-3
 - destination profiles for 2-28 to 2-29, 2-49 to 2-50
 - dialog boxes for 2-4 to 2-5, 2-9 to 2-31
 - Edit menu state for custom 2-50
 - event filter functions for 2-10 to 2-11
 - event forecasters for 2-27 to 2-28
 - event handling for the Edit menu 2-24 to 2-27, 2-43 to 2-44
 - event handling in custom 2-41 to 2-44
 - events in 2-21 to 2-28, 2-41 to 2-44
 - getting color information from 2-19 to 2-21
 - icons for custom 2-47 to 2-48
 - initialization of custom 2-37 to 2-40
 - request codes for 2-35 to 2-37
 - setting color information in 2-19 to 2-21
 - standard dialog box for 2-4 to 2-5, 2-9 to 2-12
 - user prompt strings in 2-9 to 2-10, 2-14, 2-48 to 2-49
- color profiles, response data fields A-3
- Color QuickDraw, and the Color Manager 7-3
- colors
 - color value 3-15
 - out of gamut 3-15
 - in a palette 1-6 to 1-18
 - selecting for screen depth 1-25 to 1-27
- color search functions 7-13 to 7-15
- color spaces 3-6 to 3-15
 - base families for 3-6

- CMYK 3-10 to 3-11
 - defined 3-6
 - HLS 3-8 to 3-10
 - HSV 3-8 to 3-10
 - indexed 3-14 to 3-15
 - L*a*b* 3-14
 - L*u*v* 3-14
 - RGB 3-8
 - XYZ 3-12
 - Yxy 3-13
- ColorSpec type 7-5
- ColorSync™ Profiles folder 4-6, 4-49, 6-6
- ColorSync 1.0 A-1 to A-8
 - and CMMs A-5
- ColorSync 1.0 profiles
 - and ColorSync Manager functions 2-7, A-6
 - and the CMGetProfileHeader header A-7
 - and the CMOpenProfileFile function A-7
 - and the ColorSync Manager A-5
 - contrasted with version 2.0 profiles A-3 to A-5
 - header for A-3
 - response data A-3
 - tags synthesized for A-8
- ColorSync Manager
 - and QuickDraw GX 3-26, 6-4
 - and the Color Picker Manager 2-7
 - backward compatibility A-1 to A-9
 - with ColorSync 1.0 API A-1
 - with ColorSync 1.0 profiles A-2 to A-5, A-7 to A-8
 - control panel for system profile 4-6
 - defined 3-18
 - developing CMMs 5-3 to 5-41
 - developing supportive applications 4-3 to ??
 - developing supportive device drivers 6-3 to 6-22
 - functions not supported for ColorSync 1.0 profiles A-6
 - introduction 3-3 to 3-28
 - memory allocation and use 3-19
 - picture comments for 4-15
 - programming interfaces 3-18, 4-4, 6-4
 - requirements 4-4
 - testing for availability 4-14, 6-11

- ColorSync-supportive applications
 - color-matching to a display 4-20 to 4-24
 - creating a color world for 4-27
 - creating device-linked profiles 4-53 to 4-56
 - development of 4-3 to ??
 - embedding profiles 4-34 to 4-38
 - extracting embedded profiles 4-38 to 4-49
 - features an application can implement 4-12 to 4-58
 - gamut checking 4-51 to 4-52
 - matching a bitmap 4-31
 - matching a pixel map 4-30
 - providing minimum support 4-5, 4-15
 - providing soft proofs 4-56 to 4-58
 - searching for profiles 4-49 to 4-51
- ColorSync-supportive device drivers 6-3 to 6-22
 - development of 6-10 to 6-22
 - features of 6-3
 - minimum support 6-9
 - possible features, listed 6-9 to 6-10
 - searching for profiles 6-12 to 6-15
 - setting the search criteria 6-13
 - using the search results 6-14 to 6-15
 - setting the color-matching quality flags 6-17 to 6-22
 - setting the rendering intent 6-15 to 6-17, 6-20
- color tables
 - for animation 1-12
 - default 1-18, 1-19
- ColorTable type 7-5
- color usage categories 1-8 to 1-18
 - and color allocation 1-17
 - combining 1-9, 1-16
- color values 3-15
- color worlds
 - creation of 4-27 to 4-29
 - for matching a pixel map or a bitmap 4-31
 - references for 4-29
- Commission Internationale d'Eclairage (CIE) 3-11
- complement functions 7-15
- ComponentDescription data structure 5-7
- Component Manager, and the Color Picker Manager 2-6, 2-31 to 2-40
- components (for color pickers) 2-6, 2-31 to 2-40

- concatenated profiles, creation of 4-28
- courteous colors, in palettes 1-8 to 1-9
- CProcRec type 7-15
- CreatePickerDialog function 2-18
- creating an application's default palette 1-29
- CTabChanged function 7-12
- customized dialog boxes (for color pickers) 2-5
 - to 2-6, 2-13 to 2-29
- CWConcatColorWorld function 4-55
- CWDisposeColorWorld function 4-34
- CWMatchBitmap function 4-29, 4-34
- CWMatchPixMap function 4-29, 4-32
- CWNewColorWorld function 4-28
- CWNewLinkProfile function 4-55

D

- default color tables 1-19
- default palettes 1-29
 - application 1-29, 1-30
 - obtaining 1-30
 - system 1-29
- DelComp function 7-15
- DelSearch function 7-14
- destination profile 3-24
- destination profiles, for color pickers 2-28 to
 - 2-29, 2-49 to 2-50
- device color lookup tables
 - default values 1-19
 - restored by Palette Manager 1-18
- device drivers
 - and CMMs 6-3, 6-8
 - and profiles 6-5
 - ColorSync requirements for 6-6
- device-independent color spaces. *See* CIE-based color spaces
- device-linked profiles
 - creation of 4-54 to 4-56
 - use of 4-53
- device-linked profiles request
 - defined 5-13
 - handling 5-32

- devices
 - supported by the ICC, types of 6-5
- dialog boxes (for color pickers) 2-4 to 2-5, 2-9 to
 - 2-31
- Dialog Manager, and the Color Picker
 - Manager 2-3
- direct colors 7-6
- direct devices 7-3
 - animated colors on 1-13
 - and Palette Manager 1-4
- display devices 6-5
- DoPickerEvent function 2-28

E

- Edit menu 2-24 to 2-27, 2-43 to 2-44, 2-50
- embedded profiles, support of 4-5
- event filter functions (for color pickers) 2-10 to
 - 2-11
- event forecasters 2-27 to 2-28
- event handling (in color pickers) 2-21 to 2-28
- explicit colors 1-16
 - allocation of 1-17
 - on direct devices 1-14
 - index collisions and 1-16
 - in palettes 1-8, 1-14
- explicit colors 1-14
- ExtractPickerHelpItem function 2-30

F

- format conventions xviii

G

- gamut checking 3-27
- gamuts 3-11
- GDevice type 7-4
- gestaltColorMatchingVersion selector 4-14
- GetColor function 2-4

GetNewPalette **function** 1-23
 GetPickerEditMenuState **function** 2-25
 GetPickerProfile **function** 2-29
 GetSubTable **function** 7-18
 graphics devices 7-4
 grayscale devices 1-15
 gray spaces 3-6, 3-7

H

help balloons (for color pickers) 2-29 to 2-31
 HiFi colors 3-15
 HLS space 3-8 to 3-10
 HSV space 3-8 to 3-10
 hue 3-5, 3-9

I, J

icons (for custom color pickers) 2-47 to 2-48
 index collisions (color) 1-16
 indexed color spaces 3-14 to 3-15
 indexed devices 7-3
 indexed devices, defined 1-4
 inhibited colors
 on grayscale devices 1-15
 in palettes 1-9, 1-15, 1-17
 initialization request
 defined 5-11
 handling 5-23
 input devices 6-5
 interchange color spaces 3-12
 inverse tables 7-4, 7-6 to 7-13
 ITab type 7-8

K

kDrawPicker **constant** 2-35, 2-41
 kEdit **constant** 2-35, 2-43
 kEvent **constant** 2-35, 2-41
 kExtractHelpItem **constant** 2-35

kGetColor **constant** 2-35, 2-44
 kGetDialog **constant** 2-35, 2-40
 kGetEditMenuState **constant** 2-35, 2-50
 kGetIconData **constant** 2-35, 2-47
 kGetItemList **constant** 2-35, 2-40
 kGetProfile **constant** 2-35, 2-49
 kGetPrompt **constant** 2-35, 2-48
 kInitPicker **constant** 2-35, 2-37
 kItemHit **constant** 2-35, 2-42
 kSetBaseItem **constant** 2-35
 kSetColor **constant** 2-35, 2-45
 kSetOrigin **constant** 2-35
 kSetProfile **constant** 2-35, 2-49
 kSetPrompt **constant** 2-35, 2-48
 kSetVisibility **constant** 2-35
 kTestGraphicsWorld **constant** 2-35, 2-39

L

L*a*b* space 3-14
 lightness, in HLS space 3-9
 L*u*v* space 3-14

M

Macintosh Programmer's Workshop xix
 MakeITable **function** 7-12
 matching colors, to tolerant palette requests 1-11

N

NCMBeginMatching **function** 4-21
 NCMDrawMatchedPicture **function** 4-20, 4-23
 NCMUseProfileComment **function** 4-36, A-6
 NCWNewColorWorld **function** 4-32
 new color 2-19
 NSetPalette **function**, compared to
 SetPalette 1-29

O

original color 2-19
 out-of-gamut colors 3-15
 output devices 6-5

P

Palette data type 1-5
 Palette Manager. *See also* palettes
 allocation of colors 1-5 to 1-18
 and the Color Manager 7-3, 7-16
 palette resource 1-5, 1-23
 palettes
 animated colors 1-8, 1-12 to 1-13, 1-16
 application default 1-29, 1-30
 assigning a default 1-29
 assigning to windows 1-5, 1-27
 changing and restoring for a window 1-29
 colors of 1-5
 combining usage categories 1-16
 combining usages 1-16
 courteous colors 1-8 to 1-9
 creating 1-5, 1-21 to 1-28
 creating in code 1-21 to 1-23
 data type 1-5
 default 1-29
 defined 1-5
 for different pixel depths 1-6
 on direct devices 1-4
 drawing from palette colors 1-31
 explicit colors 1-8, 1-14, 1-16
 format 1-5
 inhibited colors 1-9, 1-15, 1-16, 1-17
 resources 1-6, 1-23
 restoring a window's 1-29
 sequencing the colors 1-17
 tolerant colors 1-8, 1-11, 1-16
 usage categories 1-7 to 1-9
 perceptual matching 3-23, 6-15
 PickColor function 2-4 to 2-5, 2-8 to 2-12
 PickerDialogInfo type 2-18
 PickerIconData type 2-47

PickerMessages type 2-35
 picture comments, for the ColorSync
 Manager 4-15
 pixel depths
 default color tables for 1-19
 palettes for different depths 1-6
 pixel map color-checking request
 defined 5-12
 handling 5-31
 pixel map color-matching request
 defined 5-12
 handling 5-30
 pixel maps
 for direct devices 7-6
 for indexed devices 7-6
 'pltt' resource 1-5, 1-23
 PmBackColor function, drawing with 1-31
 pmBlack palette usage category 1-9
 PmForeColor function, drawing with 1-31
 pmWhite colors 1-5
 pmWhite palette usage category 1-9
 PostScript color rendering intent request
 handling 5-34
 PostScript color rendering request
 handling 5-34
 PostScript color rendering VM size request
 defined 5-13
 handling 5-35
 PostScript color space request
 defined 5-12
 handling 5-34
 profile flattening request
 defined 5-13
 handling 5-36, 5-37
 profile references 4-16 to 4-20
 defined 4-17
 obtaining 4-17
 profiles 3-19 to 3-22
 abstract 3-20
 and device drivers 6-5
 color space 3-20
 concatenated 4-28
 cross-platform portability 6-6
 defined 3-3, 6-5
 destination 3-24

- device 3-20
- device-linked 3-21, 3-28, 4-53 to 4-56
- device profile types 6-5
- embedded 4-5
- folder for 4-6
- format of 6-6
- locations for 4-18
- opening and obtaining a reference to 4-17
- properties of 3-21
- restrictions on searching for 6-7
- searching for 4-49 to 4-51
- source 3-24
- storage and use of 4-6, 6-6 to 6-8
- system 3-3, 4-19
- use with different device types 6-7
- vendors 6-3
- profile unflattening request, handling 5-37
- profile validation request
 - defined 5-12
 - handling 5-26
- prompts, in color pickers 2-9 to 2-10, 2-14, 2-48 to 2-49
- ProtectEntry function 7-17

Q

- quality flags 6-17 to 6-22
- QuickDraw GX, and the Color Picker Manager 2-3

R

- RealColor function 7-17
- reference white point 3-14
- relative colorimetric matching 3-23, 6-16
- rendering intents 3-23 to 3-24
 - absolute colorimetric matching 3-23
 - allowing the user to select 6-15 to 6-17
 - defined 3-23
 - perceptual matching 3-23
 - relative colorimetric matching 3-23

- saturation matching 3-23
- request codes (for color pickers) 2-35 to 2-37
- request codes for CMMs
 - optional, defined 5-25
 - required, defined 5-21
 - responding to 5-10 to 5-38
 - bitmap color checking 5-28
 - bitmap color matching 5-27
 - can do an optional request 5-22
 - closing the CMM 5-21
 - CMM version number 5-22
 - color checking 5-25
 - color matching 5-23
 - device-linked profile 5-32
 - initialization request 5-23
 - obtaining PostScript-related data 5-33 to 5-36
 - opening the CMM 5-21
 - pixel map color checking 5-31
 - pixel map color matching 5-30
 - profile flattening 5-36
 - profile unflattening 5-37
 - profile validation 5-26
 - required 5-22 to 5-25
- ReserveEntry function 7-17
- resources
 - palette 1-6, 1-23
 - 'pltt' 1-6, 1-23
 - 'thng' 5-7
 - 'thng' 2-32 to 2-33
- response data fields, for color profiles A-3
- RestoreEntries function 7-16
- RGB-based color spaces 3-7 to 3-10
 - defined 3-6, 3-7
 - HLS spaces 3-8 to 3-10
 - HSV spaces 3-8
 - RGB spaces 3-8
- RGB space 3-8

S

- sample routines
 - MyCountProfilesInPicHandle 4-41
 - MyGetIndexedProfileFromPicHandle 4-43

sample routines (*continued*)

- MyGetPrinterProfile 6-20
- MyGetSystemProfile 4-20
- MyMatchImage 4-32
- MyMatchingToDisplay 4-23
- MyOpenProfileFSSpec 4-18
- MyPrependProfileToPicHandle 4-36
- MyProfileSearch 4-50
- MyUnflattenProc 4-45

saturation 3-5, 3-9

saturation matching 3-23, 6-15

SaveEntries function 7-16

search functions 7-13 to 7-15

SetClientID function 7-14

SetEntries function 7-16, 7-17

SetPalette function 1-27

- compared to NSetPalette 1-29

SetPickerColor function 2-15, 2-16, 2-18

SetPickerProfile function 2-29

SetPickerPrompt function 2-15, 2-16, 2-18

SetPickerVisibility function 2-15, 2-16, 2-18

soft proofing 3-28

soft proofs 4-56 to 4-58, 6-7

source profile 3-24

SProcRec constant 7-14

standard dialog box (for color pickers) 2-4 to 2-5,
2-9 to 2-12

subtractive color 3-6

SystemDialogInfo type 2-15

system-owned dialog box 2-13, 2-15

system profiles

- configuring 4-21

- control panel for 4-6

- for the system display 3-3

- identifying the current system profile 4-19 to
4-20

- using quality mode and rendering intent
of 6-19

tolerant colors

- allocation of 1-17

- defined 1-11

- in palettes 1-8, 1-11

tristimulus values 3-12

U

undercolor removal 3-11

universal color spaces 3-11

usage categories

- combining 1-15, 1-16

- in palettes 1-7 to 1-9

V

value, in HSV space 3-9

W

white point 3-14

windows, palettes for 1-27

X

XYZ space 3-12

Y, Z

Yxy space 3-13

T

'thng' resources 2-32 to 2-33

tolerance, for color matching 1-11

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter Pro printer. Final page negatives were output directly from text files on an Agfa Large-Format Imagesetter. Line art was created using Adobe Illustrator™ and Adobe Photoshop™. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Adobe Letter Gothic.

Acknowledgment to Richard Collyer, Edgar Lee, David Van Brink, Wei-Ling Chu, Han Nguyen, Forrest Tanaka, John Myer, Josh Weisberg, John Wang, and to Shannon Holland, who, along with Dave Johnson, wrote the article “Pick Your Picker With Color Picker 2.0” in *develop* issue 19 on which much of the chapter “Color Picker Manager” is based.

WRITERS

Judy Melanson, Tony Francis,
Michael Kline, Rob Dearborn

DEVELOPMENTAL EDITORS

Jeanne Woodward, Beverly McGuire

ILLUSTRATORS

Bruce Lee, Ruth Anderson, Lisa Hymel

PRODUCTION EDITORS

Pat Christenson, Alan Morgenegg

PROJECT MANAGER

Trish Eastman

LEAD WRITER

Tony Francis

LEAD EDITOR

Jeanne Woodward

LEAD ILLUSTRATOR

Bruce Lee

Special thanks to David Hayward ,
Don Moccia, Steve Swen, Tom Mohr, and
Anil Gursahani.