Advanced Color Imaging Reference

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

This document was created with FrameMaker 4.0.4

 Apple Computer, Inc.
© 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.

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

No licenses, express or implied, are granted with respect to any of the technology described in this book. Apple retains all intellectual property rights associated with the technology described in this book. 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 typographical errors.

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

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

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

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.

MacPaint is a registered trademark of Claris Corporation.

NuBus is a trademark of Texas Instruments.

Motorola is a registered trademark of Motorola Corporation.

Optrotech is a trademark of Orbotech Corporation.

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.

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-nnnnn-n
1 2 3 4 5 6 7 8 9-MA-9998979695
First Printing, Month 1995

The paper used in this book meets the EPA standards for recycled fiber.

Library of Congress Cataloging-in-Publication Data

Book_title / [Apple Computer, Inc.].
p. cm.
Includes index.
ISBN 0-201-nnnnn-n
1. Macintosh (Computer)—Programming. 2.
I. Apple Computer, Inc.
nnnn.n.nnnnnnn 1995
nnn.nnn—nnnn 95-nnnnn
CIP

This document was created with FrameMaker 4.0.4

Figures, Tables, and Listings ix

Preface About This Book xi

Format of This Book and Its Companion Volume xii
Conventions Used in This Book xiv
Special Fonts xv
Types of Notes xv
Development Environment xv
For More Information xvi

Chapter 1 Palette Manager Reference 1-1

Constants and Data Types 1-3
Palette Manager Functions 1-6
Initializing the Palette Manager 1-7
Initializing and Allocating Palettes 1-8
Interacting With the Window Manager 1-11
Drawing With Color Palettes 1-16
Animating Palettes 1-20
Manipulating Palettes and Color Tables 1-23
Manipulating Palette Entries 1-27
The Palette Resource 1-30

Chapter 2 Color Picker Manager Reference 2-11

Constants and Data Structures 2-5
Gestalt Selector for the Color Picker 2-5
Picker Actions 2-5
Color Types 2-7
Edit Menu Operations 2-7
Item Hit Modifiers 2-7
Dialog Placement Specifiers 2-8
Picker Flags 2-8
Picker Attributes 2-10
Event Forecasters 2-11
Request Codes 2-12

This document was created with FrameMaker 4.0.4

Picker Color Structure 2-15
Picker Structure 2-17
Picker Icon Structure 2-17
Picker Initialization Structure 2-18
Event Filter Function 2-18
Color-Changed Function 2-19
Edit Menu Items Structure 2-19
Edit Menu State Structure 2-20
Color Picker Parameter Block 2-20
System-Owned Dialog Box Structure 2-24
Picker-Owned Dialog Box Structure 2-25
Application-Owned Dialog Box Structure 2-26
Event Data Structure 2-27
Editing Data Structure 2-29
Item Hit Structure 2-31
Help Item Structure 2-33
SmallFract Type 2-33
HSV Color Structure 2-34
HSL Color Structure 2-34
CMY Color Structure 2-35
Color Picker Manager Functions 2-36
Using the Standard Color Picker Dialog Box 2-36
Creating a Custom Color Picker Dialog Box 2-38
Handling Events in a Custom Color Picker Dialog Box 2-45
Getting Colors From and Setting Colors for a Custom Color Picker Dialog Box 2-47
Getting the Menu State and the Help Balloons for a Color Picker 2-50
Setting and Getting Color-Matching Profiles for a Color Picker 2-52
Converting Colors Among Color Models 2-54
Converting Between SmallFract and Fixed Values 2-57
Application-Defined Functions 2-58
Handling Application-Directed Events in a Color Picker 2-58
Changing Colors in a Document 2-59
Color Picker–Defined Functions 2-60
Setting Up a Color Picker 2-61
Responding to Requests to Return and Set Color Picker Information 2-66
Responding to Events in a Color Picker 2-76

Result Codes 2-80

Chapter 3 ColorSync Manager Reference for Applications and Device Drivers 3-1

The ColorSync Manager Constants and Data Structures 3-5
Constants for Profile Location Type 3-5
Constants for ColorSync Manager Gestalt Selectors and Responses 3-7
Profile Classes 3-8
Signature of the Apple-Supplied Color Management Module 3-9
Commands for Calling the Caller-Supplied ColorSync Data Transfer Functions 3-9
Picture Comment IDs for Profiles and Color Matching 3-10
Picture Comment Selectors for the cmComment ID 3-11
Color Space Signatures 3-13
Color Packing for Color Spaces 3-14
Color Spaces 3-15
Rendering Intent Values for Version 2.0 Profiles 3-19
Function Selectors for Color-Conversion-Component Functions 3-20
Operation Codes Used With PrGeneral Function 3-22
Color Conversion Component Version 3-22
The ColorSync Manager Element Tags and Their Signatures for Version 1.0 Profiles 3-22
Profile Location Union 3-23
Profile Location Structure 3-24
File Specification for a File-Based Profile 3-24
Handle Specification for a Memory-Based Profile 3-25
Pointer Specification for a Memory-Based Profile 3-25
Apple Profile Header 3-26
Profile 2.0 Header Structure for the ColorSync Manager 3-26
Concatenated Profile Set Structure 3-30
Color World Information Record 3-31
Color Management Module (CMM) Information Record Structure 3-32
Profile Search Record 3-33
XYZ Color Component Values 3-35
XYZ Color Value 3-35
Fixed XYZ Color Value 3-35

L*a*b* Color Value 3-36
L*u*v* Color Value 3-36
Yxy Color Value 3-37
RGB Color Value 3-37
HLS Color Value 3-37
HSV Color Value 3-38
CMYK Color Value 3-38
CMY Color Value 3-39
HiFi Color Values 3-39
Gray Color Value 3-39
The Color Union 3-40
The ColorSync Manager Bitmap 3-42
Profile Reference 3-43
Profile Search Result Reference 3-44
High-Level Color-Matching-Session Reference 3-44
Color World Reference 3-44
TEnableColorMatchingBlk 3-45
Profile Header for ColorSync 1.0 3-45
PostScript Color Rendering Dictionary (CRD) Virtual Memory Size Tag Structure 3-48
The ColorSync Manager Functions 3-49
Accessing Profile Files 3-50
Accessing Profile Elements 3-60
Matching Colors Using the High-Level Functions 3-75
Using Embedded Profiles With QuickDraw 3-78
Matching Colors Using the Low-Level Functions Without QuickDraw 3-80
Assigning and Accessing the System Profile File 3-99
Searching External Profiles 3-101
Converting Between Color Spaces 3-106
PostScript Color-Matching Support Functions 3-125
Locating the ColorSync Profiles Folder 3-130
Application-Defined Functions for the ColorSync Manager 3-131
Result Codes 3-138

Chapter 4 ColorSync Manager Reference for Color Management Modules 4-1

Constants 4-3
Color Management Module Component Interface 4-3
Required Request Codes 4-4
Optional Request Codes 4-5
Required Functions 4-9
Optional Functions 4-14

Chapter 5 Color Manager Reference 5-1

Constants and Data Types 5-3
Color Manager Functions 5-5
Managing Colors 5-5
Managing Color Tables 5-10
Operations on Search and Complement Functions 5-15
Application-Defined Functions 5-17

Glossary GL-1

Index IN-1

Figures, Tables, and Listings

Preface About This Book xi

Figure P-1 Road map to *Advanced Color Imaging* xiv
Figure 1-1 Format of a palette resource 1-31

This document was created with FrameMaker 4.0.4

About This Book

The *Advanced Color Imaging Reference*, and its companion, *Advanced Color Imaging on the Mac OS*, 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 provide a reference to use these managers, which you can use 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

This document was created with FrameMaker 4.0.4

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. However, for completeness, the functions and data structures of the Color Manager are described in this book.

Format of This Book and Its Companion Volume

This book provides a reference chapter for the Palette Manager, the Color Picker Manager, the Color Manager, and two chapters for the ColorSync Manager (one chapter describing routines to develop ColorSync applications and device drivers and the other describing routines and request codes that allow color management modules to respond to ColorSync-supportive applications). 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 routine.

The book *Advanced Color Imaging on the Mac OS* provides conceptual information about enhancing your application's color capabilities; it also includes code samples with 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.

In the same book, 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, routines, and result codes associated with the Color Picker Manager.

The *Advanced Color Imaging Reference* comes is in an electronic form only—there is no printed version of it. It has two online 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.

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 electronic version is in Acrobat format.

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

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

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

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 LetterGothic (`this is LetterGothic`).

Words that appear in **boldface** are key terms or concepts and are defined in the glossary at the end of this book. Note that numerical entries (for example, **32-bit clean**) are sorted before all alphabetical entries in the glossary and in the index.

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 1-8.) ◆

IMPORTANT
A note like this contains information that is essential for an understanding of the main text. (An example appears on page 1-13.) ▲

▲ **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 5-8.) ▲

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.

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

Palette Manager Reference

Contents

Constants and Data Types 1-3
`Usage Constants` 1-3
`Update Constants` 1-4
`The Palette Structure` 1-5
`The Color Information Structure` 1-5
Palette Manager Functions 1-6
Initializing the Palette Manager 1-7
`InitPalettes` 1-7
`PMgrVersion` 1-7
Initializing and Allocating Palettes 1-8
`GetNewPalette` 1-8
`NewPalette` 1-9
`DisposePalette` 1-10
Interacting With the Window Manager 1-11
`SetPalette` 1-11
`NSetPalette` 1-12
`ActivatePalette` 1-13
`GetPalette` 1-14
`SetPaletteUpdates` 1-15
`GetPaletteUpdates` 1-15
Drawing With Color Palettes 1-16
`PmForeColor` 1-16
`PmBackColor` 1-17
`SaveFore` 1-17
`RestoreFore` 1-18
`SaveBack` 1-19
`RestoreBack` 1-20

This document was created with FrameMaker 4.0.4

Animating Palettes 1-20
`AnimateEntry` 1-21
`AnimatePalette` 1-22
Manipulating Palettes and Color Tables 1-23
`CopyPalette` 1-23
`ResizePalette` 1-24
`RestoreDeviceClut` 1-24
`CTab2Palette` 1-25
`Palette2CTab` 1-26
Manipulating Palette Entries 1-27
`GetEntryColor` 1-27
`GetEntryUsage` 1-27
`SetEntryColor` 1-28
`SetEntryUsage` 1-29
`Entry2Index` 1-30
The Palette Resource 1-30

This reference document describes the data types, constants, functions, and resource that are specific to the Palette Manager.

The “Constants and Data Types” section shows the usage constants enumeration and the `Palette` and `ColorInfo` data types.

The “Palette Manager Functions” section describes functions for initializing, manipulating, and allocating palettes; drawing with palette colors; and interacting with the Window Manager.

The “Resource” section contains a description of the palette resource, from which palette structures can be made.

Constants and Data Types

This section describes the usage constants enumeration, which allows you to assign usage categories to each color in a palette, and the update constants enumeration, which allows you to specify the conditions under which a window’s color environment is updated. It also describes the palette structure, which contains the collection of colors you create for your application and information about their use; and the color information structure, which is the part of the palette structure that contains the information about a particular color.

Usage Constants

The usage constants define how each color in a palette is to be used. (Note that you can combine certain of these constants. See “Colors in a Palette” in the chapter “Palette Manager” in the book *Advanced Color Imaging on the Mac OS* for information on how to use these constants to specify the usage for the colors in a palette.)

The `ciUsage` field of the color information structure (described on page 1-5) contains one or more usage constants that define how a particular color in a palette is to be used.

```
/* usage contsants */
enum {
        pmCourteous     = $0000;    /* courteous color */
```


This document was created with FrameMaker 4.0.4

```
        pmTolerant      = $0002;    /* tolerant color */
        pmAnimated      = $0004;    /* animated color */
        pmExplicit      = $0008;    /* explicit color */
        pmWhite         = $0010;    /* use on 1-bit device */
        pmBlack         = $0020;    /* use on 1-bit device */
        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 */
};
```

Update Constants

The update constants determine whether a window is updated based on various changes to the color environment. You use the update constants with the `nCUpdates` parameter of the `NSetPalette` function (described on page 1-12) and the `updates` parameter of the `SetPaletteUpdates` function (described on page 1-15).

```
/* update constants */
enum {
pmNoUpdates = $8000
pmBkUpdates = $A000
pmFgUpdates = $C000
pmAllUpdates = $E000
};
```

Constant descriptions

`pmNoUpdates = $8000`
Do not update the window when its color environment changes.

`pmBkUpdates = $A000`
Update the window only when it is *not* the active window.

`pmFgUpdates = $C000`
Update the window only when it is the active window.

`pmAllUpdates = $E000`
Update the window whenever its color environment changes.

The Palette Structure

A palette structure contains a header and a collection of color information structures, one for each color in the palette. The `Palette` data type defines a palette structure.

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

Field descriptions

`pmEntries`	The number of `ColorInfo` structures in the `pmInfo` array.
`pmDataFields`	Private fields used by the Palette Manager.
`pmInfo`	An array of `ColorInfo` structures, described next.

The Color Information Structure

Each color information structure in a palette comprises an RGB color value, information describing how the color is to be used, a tolerance value for colors that need only be approximated, and private fields. You should not create and modify the public fields directly; instead, use Palette Manager functions such as `SetEntryColor` and `SetEntryUsage`.

The `ColorInfo` data type defines a color information structure.

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

Field descriptions

`ciRGB` An RGB color value, which is defined by the `RGBColor` structure (see the chapter "Color QuickDraw" in *Inside Macintosh: Imaging With QuickDraw*). It contains three fields that contain integer values for defining, respectively, the red, green, and blue values of the color.

`ciUsage` One or more of the usage constants, specifying how this entry is to be used. The `ciUsage` field can contain any of the usage constants, which are listed in "Usage Constants" on page 1-3.

`ciTolerance` An integer expressing the range in RGB space within which the red, green, and blue values must fall to satisfy this entry. A tolerance value of $0000 means that only an exact match is acceptable. Values of $0*xxx* other than $0000 are reserved and should not be used in applications.

`ciDataFields` Private fields.

Palette Manager Functions

This section describes Palette Manager functions for initializing the Palette Manager, initializing and allocating palettes, interacting with the Window Manager, drawing with color palettes, animating palettes, and manipulating palettes, color tables, and palette entries.

The functions `SaveFore`, `RestoreFore`, `SaveBack`, `RestoreBack`, `ResizePalette`, and `RestoreDeviceClut` are available only with system software versions 6.0.5 and later and with the 32-Bit QuickDraw system extension.

Initializing the Palette Manager

This section describes functions that initialize the Palette Manager and determine the version of the Palette Manager that is running.

InitPalettes

The `InitPalettes` function initializes the Palette Manager.

```
pascal void InitPalettes(void);
```

DESCRIPTION

The `InitPalettes` function searches for devices that support a device color table and initializes an internal data structure for each one. Your application does not have to call `InitPalettes` because the Window Manager's `InitWindows` function calls it automatically.

PMgrVersion

Use the `PMgrVersion` function to determine which version of the Palette Manager is executing; it returns an integer specifying the version number.

```
pascal short PMgrVersion(void);
```

DESCRIPTION

The values that the `PMgrVersion` function may return and their meaning are as follows:

Value	Description
$0202	System software version 7.0
$0201	System software version 6.0.5
$0200	Original 32-Bit QuickDraw system extension

Initializing and Allocating Palettes

This section describes functions for creating and disposing of palettes. You can create a new palette from a 'pltt' resource using the GetNewPalette function or create a palette from within your application using the NewPalette function. You can also let the Palette Manager and Window Manager together create a palette by creating a 'pltt' resource with the same ID as the window you want to assign it to.

Use the DisposePalette function to dispose of an entire palette.

GetNewPalette

Use the GetNewPalette function to create and initialize a palette from a 'pltt' resource (described on page 1-30).

```
pascal PaletteHandle GetNewPalette(short PaletteID);
```

paletteID The resource ID of the source palette.

DESCRIPTION

The GetNewPalette function creates a palette from information supplied by the palette resource specified in the paletteID parameter; it also initializes the new palette.

Note
The GetNewPalette function detaches the resource when it creates the new palette, so you do not need to call the ReleaseResource function. ◆

If you open a new color window with GetNewCWindow, the Window Manager calls GetNewPalette automatically, with paletteID equal to the window's resource ID. Therefore, if you have created a palette resource with the same ID as a window, the Window Manager and Palette Manager automatically create the palette for you and your application needn't call GetNewPalette to create the palette.

SEE ALSO

To attach a palette to a window after creating it, use the `SetPalette` function, described on page 1-11.

To change the entries in a palette after creating it, use the `SetEntryColor` (page 1-28) and the `SetEntryUsage` (page 1-29) functions.

NewPalette

Use the `NewPalette` function to allocate a new palette from colors in the color table.

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

`entries` The number of `ColorInfo` structures to be created in the new palette.

`srcColors` The color table from which the colors are to be obtained.

`srcUsage` The usage value to be assigned each `ColorInfo` structure in the palette.

`srcTolerance`
The tolerance value to be assigned each `ColorInfo` structure in the palette.

DESCRIPTION

The `NewPalette` function fills the palette with as many RGB values from the color table as it has or can fit. `NewPalette` sets the `usage` field of each color to the value in the `srcUsage` parameter and the tolerance value of each color to the value in the `srcTolerance` parameter. If no color table is provided (`srcColors` = `nil`), then all colors in the palette are set to black (red, green, and blue equal to $0000).

SEE ALSO

For an example of using the `NewPalette` function to create a palette, see Listing 1-1 in the chapter "Palette Manager" in the book *Advanced Color Imaging on the Mac OS.*

To attach a palette to a window after creating it, use the `SetPalette` function, described on page 1-11.

To change the entries in a palette after creating it, use the `SetEntryColor` (page 1-28) and the `SetEntryUsage` (page 1-29) functions.

DisposePalette

Use the `DisposePalette` function to dispose of a palette.

```
pascal void DisposePalette(PaletteHandle srcPalette);
```

`srcPalette` A handle to the palette to be disposed of.

DESCRIPTION

The `DisposePalette` function disposes of the palette you specify in the `srcPalette` parameter. If the palette has any entries allocated for animation on any screen device, then `DisposePalette` relinquishes these entries before the palette's memory is released.

If a palette is attached to a window automatically—because the palette resource and the window have the same ID—you do not have to call the `DisposePalette` function to dispose of the function. The Palette Manager and Window Manager dispose of the palette automatically if the palette is replaced or if the window goes away.

However, if you explicitly attach a palette to a window with the `SetPalette` or `NSetPalette` function, your application owns the palette and is responsible for disposing of it.

SPECIAL CONSIDERATIONS

It is possible to attach a single palette to multiple windows; therefore, even when a window goes away and no longer needs a palette, other windows may still need it.

Interacting With the Window Manager

This section describes functions that interact with the Window Manager. You can use both the `SetPalette` and `NSetPalette` functions to attach a palette to a window.

You use the `ActivatePalette` function to adjust the color environment whenever your window's status changes or after making changes to a palette. You can use the `GetPalette` function to return a handle to the palette currently associated with a specified window. Use the `SetPaletteUpdates` and `GetPaletteUpdates` functions to explicitly set and get the update conditions for a palette.

SetPalette

Use the `SetPalette` function to associate a palette with a window.

```
pascal void SetPalette(WindowPtr dstWindow,
                       PaletteHandle srcPalette,
                       Boolean cUpdates);
```

`dstWindow` A pointer to the window to which you want to assign a new palette.

`srcPalette` A handle to the palette you want to assign.

`cUpdates` A Boolean value in which you specify whether the window is to receive updates as a result of changes to the color environment. If you want the window to be updated whenever its color environment changes, set the `cUpdates` parameter to `TRUE`.

SPECIAL CONSIDERATIONS

The `cUpdates` parameter controls whether changes to the color environment cause update events to be sent to the specified window only if the window is *not* the frontmost window. When a window is the frontmost window, changes to its palette cause it to get an update event regardless of how the `cUpdates` parameter is set. You can use the `NSetPalette` function, which does the same thing as `SetPalette`, when you need greater flexibility in setting criteria for updates. The `nCUpdates` parameter for the `NSetPalette` function includes the option of turning off updates when the window is the frontmost window. ◆

SEE ALSO

For an example of using the `SetPalette` function to attach a palette to a window, see Listing 1-4 on page 1-27 of the book *Advanced Color Imaging on the Mac OS.*

Use the `NSetPalette` function (described on page 1-12) to associate a palette with a window but with additional options as to when an update event is triggered by changes to the color environment.

Use the `GetNewPalette` function (described on page 1-8) or the `NewPalette` function (described on page 1-9) to create a new palette.

To dispose of a palette, use the `DisposePalette` function, described on page 1-10.

NSetPalette

You can use the `NSetPalette` function to associate a new palette with a window; it is identical to the `SetPalette` function (described on page 1-11) except that the `nCUpdates` parameter is an integer rather than a Boolean value, so that a variety of conditions can trigger an update event.

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

`dstWindow` A pointer to the window to which you want to assign a new palette.

`srcPalette` A pointer to the palette you want to assign.

`nCUpdates` An integer value in which you specify whether the window is to receive updates as a result of various changes to the color environment. See "Update Constants" on page 1-4 for a description of the update options.

DESCRIPTION

`NSetPalette` changes the palette associated with the window specified in the `dstWindow` parameter to the palette specified by `srcPalette`. `NSetPalette` also records whether the window is to receive updates as a result of changes to its color environment. The update constants, which you pass to the `nCUpdates` parameter, determine when the window is updated.

IMPORTANT

The `NSetPalette` function is available in system software versions 6.0.2 and later. ▲

SEE ALSO

Use the `SetPalette` function (described on page 1-11) if you don't need the flexibility that `NSetPalette` provides for update events.

Use the `GetNewPalette` function (described on page 1-8) or the `NewPalette` function (described on page 1-9) to create a new palette.

To dispose of a palette, use the `DisposePalette` function, described on page 1-10.

ActivatePalette

The `ActivatePalette` function compares the color environment with the color requirements of your window; it then changes the device color tables and generates window updates as needed.

```
pascal void ActivatePalette(WindowPtr srcWindow);
```

`srcWindow` A pointer to the window for which you want status changes reported.

DESCRIPTION

The Window Manager calls ActivatePalette when your window's status changes—for example, when your window opens, closes, moves, or becomes frontmost. You need to call the ActivatePalette function yourself if you change a palette—for example, by changing a color with the SetEntryColor function—and you want the changes to take place immediately, before the Window Manager would do it.

If the window specified in the srcWindow parameter is frontmost, ActivatePalette examines the information stored in the window's palette and attempts to provide the color environment described therein. It determines a list of devices on which to render the palette by intersecting the port rectangle of the window with each device. If the intersection is not empty and if the device has a color table, then ActivatePalette checks to see if the color environment is sufficient. If a change is required, ActivatePalette calls the Color Manager to reserve or modify the device's color entries as needed. The ActivatePalette function then generates update events for all windows that need color updates.

Calling ActivatePalette with an offscreen graphics world has no effect.

GetPalette

Use the GetPalette function to obtain a window's palette.

```
pascal PaletteHandle GetPalette(WindowPtr srcWindow);
```

srcWindow A pointer to the window for which you want the associated palette.

DESCRIPTION

The GetPalette function returns a handle to the palette associated with the window specified in the srcWindow parameter. If the window has no associated palette or if the window is not a color window, the GetPalette function returns nil.

Normally, the GetPalette function does not allocate memory, with one exception. When your application calls GetPalette to get a copy of the default application palette, the Palette Manager looks at the AppPalette global variable.

If `AppPalette` is `nil`, `GetPalette` makes a copy of the default system palette and returns this copy. In all other cases, `GetPalette` returns a handle to the requested palette.

You request the default palette as follows:

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

SEE ALSO

For more information about the default application palette, see "Designating a Default Palette for Your Application" on page 1-28 of the book *Advanced Color Imaging on the Mac OS*.

SetPaletteUpdates

Use the `SetPaletteUpdates` function to set the update attribute of a palette.

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

`p` A handle to the palette.

`updates` One of the update attributes for the `NSetPalette` function. See "Update Constants" on page 1-4 for a description of the update attributes.

GetPaletteUpdates

Use the `GetPaletteUpdates` function to obtain the update attribute of a palette.

```
pascal short GetPaletteUpdates(PaletteHandle p);
```

`p` A handle to the palette.

DESCRIPTION

The `GetPaletteUpdates` function returns one of the update attributes described in "Update Constants" on page 1-4.

Drawing With Color Palettes

This section describes the functions that you can use to draw with. You can use the `PmForeColor` and `PmBackColor` functions to specify foreground and background drawing colors with the assistance of the Palette Manager.

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

PmForeColor

Use the `PmForeColor` function to set the foreground color field of the current graphics port to a palette color.

```
pascal void PmForeColor(short dstEntry);
```

`dstEntry` The palette entry whose color is to be used as the foreground color.

DESCRIPTION

The `PmForeColor` function sets the current color graphics port's `rgbFgColor` field to match the color in the entry specified by the `dstEntry` parameter of the palette associated with the current window structure. 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.

SEE ALSO

The color graphics port is described in the chapter "Color QuickDraw" in *Inside Macintosh: Imaging With QuickDraw.*

PmBackColor

Use the `PmBackColor` function to set the background color field of the current graphics port to a palette color.

```
pascal void PmBackColor(short dstEntry);
```

`dstEntry` The palette entry whose color is to be used as the background color.

DESCRIPTION

The `PmBackColor` function sets the current color graphics port's `rgbBkColor` field to match the color in the entry specified by the `dstEntry` parameter of the palette associated with the current window structure. For courteous and tolerant entries, `PmBackColor` calls the `RGBBackColor` function using the RGB color of the palette entry. For animated colors, `PmBackColor` selects the recorded device index previously reserved for animation (if still present) and installs it in the color graphics port. The `rgbBgColor` field is set to the value from the palette entry. For explicit colors, `PmBackColor` 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, ***maxIndex*** varies appropriately for each device.

SaveFore

Use the `SaveFore` function to save the current foreground color.

```
pascal void SaveFore(ColorSpec *c);
```

`c` A `ColorSpec` structure (described in the chapter "Color QuickDraw" of *Inside Macintosh: Imaging With QuickDraw*) to hold the current foreground color.

DESCRIPTION

The `SaveFore` function returns the current foreground color in the `ColorSpec` structure specified in the `c` parameter. You can save either Color QuickDraw's foreground color from the `CGrafPort` structure or the Palette Manager's foreground color from the `GrafVars` structure. A value of 0 in the `value` field of the `ColorSpec` structure specifies retrieving the RGB color from the `rgbFgColor` field of the `CGrafPort` structure; a value of 1 in the `value` field specifies retrieving the palette entry from the `pmFgColor` field of the `GrafVars` structure.

IMPORTANT

The `SaveFore` function is available only with system software versions 6.0.5 and later and with the 32-Bit QuickDraw system extension. ▲

RestoreFore

Use the `RestoreFore` function to set the current foreground color to the color you supply.

```
pascal void RestoreFore(const ColorSpec *c);
```

`c` A `ColorSpec` structure (described in the chapter "Color QuickDraw" of *Inside Macintosh: Imaging With QuickDraw*) containing the RGB color to be set as the foreground color.

DESCRIPTION

The `RestoreFore` function stores the RGB color of the `ColorSpec` structure you specify by the `c` parameter as the current foreground color. You can store the color into either Color QuickDraw's foreground color in the `CGrafPort` structure or the Palette Manager's foreground color in the `GrafVars` structure. If you specify 0 in the `value` field of the `ColorSpec` structure, the `RestoreFore` function stores the RGB value in the `rgbFgColor` field of the current `CGrafPort`

structure. If you specify 1 in the `value` field of the `ColorSpec` structure, the `RestoreFore` function stores the RGB value in the `pmFgColor` field of the `GrafVars` structure.

IMPORTANT

The `RestoreFore` function is available only with system software versions 6.0.5 and later and with the 32-Bit QuickDraw system extension. ▲

SaveBack

Use the `SaveBack` function to save the current background color.

```
pascal void SaveBack(ColorSpec *c);
```

`c` A `ColorSpec` structure (described in the chapter "Color QuickDraw" of *Inside Macintosh: Imaging With QuickDraw*) to hold the current background color.

DESCRIPTION

The `SaveBack` function returns the current background color in the `c` parameter. You can save either Color QuickDraw's background color from the `CGrafPort` structure or the Palette Manager's background color from the `GrafVars` structure. A value of 0 in the `value` field of the `ColorSpec` structure specifies retrieving the RGB color from the `rgbBkColor` field of the `CGrafPort` structure; a value of 1 in the `value` field specifies retrieving the palette entry from the `pmBkColor` field of the `GrafVars` structure.

IMPORTANT

The `SaveBack` function is available only with system software versions 6.0.5 and later and with the 32-Bit QuickDraw system extension. ▲

RestoreBack

Use the `RestoreBack` function to set the current background color to the color you specify.

```
pascal void RestoreBack(const ColorSpec *c);
```

`c` A `ColorSpec` structure (described in the chapter "Color QuickDraw" of *Inside Macintosh: Imaging With QuickDraw*) containing the RGB color to be set as the background color.

DESCRIPTION

The `RestoreBack` function stores the RGB color of the `ColorSpec` structure specified by the `c` parameter as the current background color. You can restore either Color QuickDraw's foreground color in the `CGrafPort` structure or the Palette Manager's background color in the `GrafVars` structure. If you specify 0 in the `value` field of the `ColorSpec` structure, the `RestoreBack` function stores the RGB value in the `rgbFgColor` field of the current `CGrafPort` structure. If you specify 1 in the `value` field of the `ColorSpec` structure, the `RestoreBack` function stores the RGB value in the `pmBkColor` field of the `GrafVars` structure.

IMPORTANT

The `RestoreBack` function is available only with system software versions 6.0.5 and later and with the 32-Bit QuickDraw system extension. ▲

Animating Palettes

To use color-table animation, you can change the colors in a palette and on corresponding devices with the `AnimateEntry` and `AnimatePalette` functions.

AnimateEntry

Use the `AnimateEntry` function to change the color of a window's palette entry.

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

`dstWindow` A pointer to the window whose palette color is to be changed.

`dstEntry` The palette entry to be changed.

`srcRGB` The new RGB value.

DESCRIPTION

The `AnimateEntry` function changes the RGB value of an animated entry for a window's palette. Each device for which that index has been reserved is immediately modified to contain the new value. This is not considered to be a change to the device's color environment because no other windows should be using the animated entry.

If the palette entry is not an animated color or if the associated indexes are no longer reserved, no animation occurs.

If you have blocked color updates in a window by using `SetPalette` with `cUpdates` set to `FALSE`, you may observe unintentional animation. This occurs when `ActivatePalette` reserves for animation device indexes that are already used in the window. Redrawing the window, which normally is the result of a color update event, removes any animated colors that do not belong to the window.

SEE ALSO

For an example of using the `AnimateEntry` function to achieve color animation effects, see Listing 1-5 on page 1-31 of the book *Advanced Color Imaging on the Mac OS*.

AnimatePalette

Use the AnimatePalette function to change the colors of a series of palette entries; it is similar to the AnimateEntry function, but it acts upon a range of entries.

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

dstWindow — A pointer to the window whose palette colors are to be changed.

srcCTab — A handle to the color table containing the new colors. Color tables are described in the chapter "Color QuickDraw" of *Inside Macintosh: Imaging With QuickDraw.*

srcIndex — The source color table entry at which copying starts.

dstEntry — The palette entry at which copying starts.

dstLength — The number of palette entries to be changed.

DESCRIPTION

The AnimatePalette function changes the colors of a series of palette entries. Beginning at the index specified by the srcIndex parameter (which has a minimum value of 0), the number of entries specified in dstLength are copied from the source color table to the destination window's palette, beginning at the entry specified in the dstEntry parameter. If the source color table specified in srcCTab is not sufficiently large to accommodate the request, AnimatePalette modifies as many entries as possible and leaves the remaining entries unchanged.

SEE ALSO

For an example of using the AnimatePalette function to achieve color animation effects, see Listing 1-5 on page 1-31 of the book *Advanced Color Imaging on the Mac OS.*

Manipulating Palettes and Color Tables

You can use the `CopyPalette` function to copy palettes from other palettes and from color tables, and you can use the `ResizePalette` function to resize palettes. The `RestoreDeviceClut` function restores the color table of a device to its default set of colors. `CTab2Palette` copies the specified color table into a palette, and its opposite, `Palette2CTab`, copies a palette into a color table. Each function resizes the target object as needed.

CopyPalette

Use the `CopyPalette` function to copy entries from one palette to another.

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

`srcPalette` A handle to the palette from which colors are copied.

`dstPalette` A handle to the palette to which colors are copied.

`srcEntry` The source palette entry at which copying starts.

`dstEntry` The destination palette entry at which copying starts.

`dstLength` The number of destination palette entries to change.

DESCRIPTION

The `CopyPalette` function copies entries from the source palette into the destination palette. The copy operation begins at the values specified by the `srcEntry` and `dstEntry` parameters, copying into as many entries as are specified by the `dstLength` parameter. `CopyPalette` resizes the destination palette when the number of entries after the copy operation is greater than it was before the copy operation.

`CopyPalette` does not call `ActivatePalette`, so your application is free to change the palette a number of times without causing a series of intermediate changes

to the color environment. Your application should call `ActivatePalette` after completing all palette changes.

If either of the palette handles is `nil`, `CopyPalette` does nothing.

ResizePalette

Use the `ResizePalette` function to change the size of a palette.

```
pascal void ResizePalette(PaletteHandle srcPalette, short size);
```

`srcPalette` A handle to the palette to be resized.

`size` The number of resulting entries in the palette.

DESCRIPTION

The `ResizePalette` function sets the palette specified in `srcPalette` to the number of entries indicated in the `size` parameter. If `ResizePalette` adds entries at the end of the palette, it sets them to `pmCourteous`, with the RGB values set to (0,0,0)—that is, black. If `ResizePalette` deletes entries from the end of the palette, it safely disposes of them.

RestoreDeviceClut

Use the `RestoreDeviceClut` function to set the color table of a graphics device to its default state.

```
pascal void RestoreDeviceClut(GDHandle gdh);
```

`gdh` A handle to the `GDevice` structure (described in the chapter "Graphics Devices" of *Inside Macintosh: Imaging With QuickDraw*) to be restored.

DESCRIPTION

The `RestoreDeviceClut` function changes the color table of the device specified by the `gdh` parameter to its default state. If this process changes any entries, the Palette Manager posts color updates to windows intersecting the device. Pass `nil` in the `gdh` parameter to restore all screens.

You don't need to use this function to restore the Finder's desktop colors, because its colors are automatically restored upon switching from applications that use the Palette Manager. Likewise, you needn't worry when switching to another application that uses the Palette Manager. Although colors are not automatically restored in this case, if that application needs a certain set of colors, the Palette Manager provides them the moment that application comes to the front.

The reason to use `RestoreDeviceClut` is that you may be switching to an application that does not use the Palette Manager, in which case that application inherits your palette unless you restore the default color lookup tables for all the available display devices.

CTab2Palette

Use the `CTab2Palette` function to copy the colors of a color table into a palette.

```
pascal void CTab2Palette(CTabHandle srcCTab,
                        PaletteHandle dstPalette,
                        short srcUsage,
                        short srcTolerance);
```

`srcCTab` A handle to the color table whose colors are to be copied. Color tables are described in the chapter "Color QuickDraw" of *Inside Macintosh: Imaging With QuickDraw.*

`dstPalette` The palette to receive the colors.

`srcUsage` A usage constant to be assigned the palette entries. Usage constants are described in "Usage Constants" on page 1-3.

`srcTolerance` A tolerance value to be assigned the palette entries.

DESCRIPTION

The `CTab2Palette` function copies the fields from an existing color-table structure into an existing palette structure. If the structures are not the same size, then `CTab2Palette` resizes the palette structure to match the number of entries in the color-table structure. If the palette in `dstPalette` has any entries allocated for animation on any screen device, they are relinquished before the new colors are copied. The `srcUsage` and `srcTolerance` parameters are the value that you assign to the new colors.

If you want to use color-table animation, you can use `AnimateEntry` (described on page 1-21) and `AnimatePalette` (described on page 1-22) to change the colors in a palette and on corresponding devices. Changes made to a palette by `CTab2Palette` don't take effect until the next `ActivatePalette` function is performed. If either the color-table handle or the palette handle is `nil`, `CTab2Palette` does nothing.

Palette2CTab

Use the `Palette2CTab` function to copy the colors of a palette into a color table.

```
pascal void Palette2CTab(PaletteHandle srcPalette,
                         CTabHandle dstCTab);
```

`srcPalette` A handle to the palette whose colors are to be used.

`dstCTab` A handle to the color table to receive the colors. Color tables are described in the chapter "Color QuickDraw" of *Inside Macintosh: Imaging With QuickDraw.*

DESCRIPTION

The `Palette2CTab` function copies all of the colors from an existing palette structure into an existing color-table structure. If the structures are not the same size, then `Palette2CTab` resizes the color-table structure to match the number of entries in the palette structure. If either the palette handle or the color-table handle is `nil`, `Palette2CTab` does nothing.

Manipulating Palette Entries

The `GetEntryColor`, `GetEntryUsage`, `SetEntryColor`, and `SetEntryUsage` functions allow your application to retrieve and modify the fields of a palette. The `Entry2Index` function returns an index for a palette entry.

GetEntryColor

Use the `GetEntryColor` function to obtain the color of a palette entry.

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

`srcPalette` A handle to the palette to be accessed.

`srcEntry` The palette entry whose color is desired.

`dstRGB` An RGB color structure to receive the palette color. RGB color structures are described in the chapter "Color QuickDraw" of *Inside Macintosh: Imaging With QuickDraw.*

DESCRIPTION

The `GetEntryColor` function takes the RGB color of the entry specified by the `srcEntry` parameter and stores it in the destination RGB color structure. You can modify the entry's color using the `SetEntryColor` function.

GetEntryUsage

Use the `GetEntryUsage` function to obtain the usage and tolerance fields of a palette entry.

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

`srcPalette` A handle to the palette to be accessed.

`srcEntry` The palette entry whose usage and tolerance are desired.

`dstUsage` The usage value of the palette entry.

`dstTolerance`
The tolerance value of the palette entry.

DESCRIPTION

The `GetEntryUsage` function takes the usage and tolerance values of the entry specified by the `srcEntry` parameter and stores them in the `dstUsage` and `dstTolerance` parameters. You can modify the entry's usage and tolerance values by using the `SetEntryUsage` function.

SetEntryColor

Use the `SetEntryColor` function to change the color of a palette entry.

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

`dstPalette` The palette whose entry color is to be changed.

`dstEntry` The palette entry to be changed.

`srcRGB` The new RGB color value.

DESCRIPTION

The `SetEntryColor` function stores the RGB color of the `srcRGB` parameter in the palette entry specified by the `dstEntry` parameter. `SetEntryColor` marks the entry as having changed, but it does not change the color environment. The change occurs upon the next call to `ActivatePalette`. `SetEntryColor` marks modified entries such that the palette is updated, even though no update is required by a change in the color environment.

SEE ALSO

For an example of using the `SetEntryColor` function to change the colors in a palette, see Listing 1-1 on page 1-21 of the book *Advanced Color Imaging on the Mac OS*.

SetEntryUsage

Use the `SetEntryUsage` function to modify the usage category and tolerance values of a palette entry.

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

`dstPalette` A handle to the palette to be modified.

`dstEntry` The palette entry.

`srcUsage` The new usage value; one or more usage constants.

`srcTolerance`
The new tolerance value.

DESCRIPTION

The `SetEntryUsage` function stores the usage and tolerance values specified by the `srcUsage` and `srcTolerance` parameters into the palette entry specified by the `dstEntry` parameter. `SetEntryUsage` marks the entry as having changed, but it does not change the color environment. The change occurs upon the next call to `ActivatePalette`. Modified entries are marked such that the palette is updated even though no update is required by a change in the color environment. If either `srcUsage` or `srcTolerance` is set to $FFFF (–1), the entries are not changed.

This function allows you to easily modify a palette created with `NewPalette` or modified by `CTab2Palette`. For such palettes the `ciUsage` and `ciTolerance` fields of the `ColorInfo` structure are the same because you can designate only one value for each. You typically call `SetEntryUsage` after `NewPalette` or `CTab2Palette` to adjust and customize your palette.

SEE ALSO

For an example of using the `SetEntryUsage` function to change the usage and tolerance of a color in a palette, see Listing 1-3 on page 1-24 of the book *Advanced Color Imaging on the Mac OS*.

Entry2Index

Use the `Entry2Index` function to obtain the index for a specified entry in the current graphics port's palette on the current device.

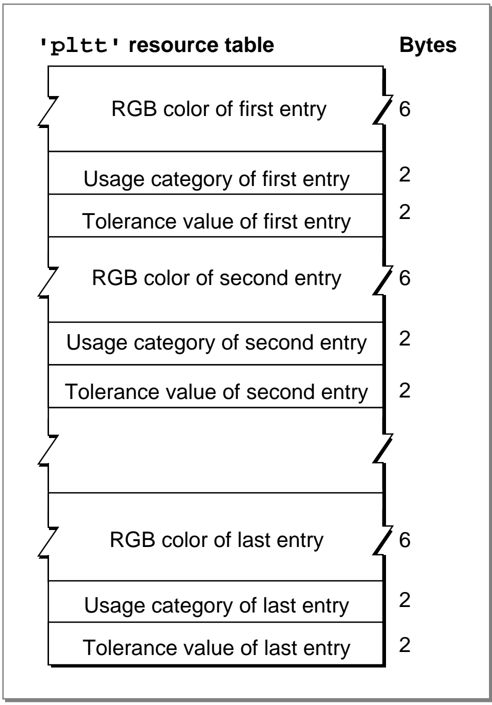
```
pascal long Entry2Index(short entry);
```

`entry` The palette entry whose equivalent device index is to be returned.

The Palette Resource

The palette resource contains the color values and the usage and tolerance constants; in effect, it is a series of `ColorInfo` structures without the private fields. The Palette Manager adds its private fields both to the header and to each `ColorInfo` structure when it creates a palette structure from the `'pltt'` resource. The format of a palette resource is shown in Figure 1-1.

Figure 1-1 Format of a palette resource



Color Picker Manager Reference

Contents

Constants and Data Structures 2-5
- Gestalt Selector for the Color Picker 2-5
- Picker Actions 2-5
- Color Types 2-7
- Edit Menu Operations 2-7
- Item Hit Modifiers 2-7
- Dialog Placement Specifiers 2-8
- Picker Flags 2-8
- Picker Attributes 2-10
- Event Forecasters 2-11
- Request Codes 2-12
- Picker Color Structure 2-15
- Picker Structure 2-17
- Picker Icon Structure 2-17
- Picker Initialization Structure 2-18
- Event Filter Function 2-18
- Color-Changed Function 2-19
- Edit Menu Items Structure 2-19
- Edit Menu State Structure 2-20
- Color Picker Parameter Block 2-20
- System-Owned Dialog Box Structure 2-24
- Picker-Owned Dialog Box Structure 2-25
- Application-Owned Dialog Box Structure 2-26
- Event Data Structure 2-27
- Editing Data Structure 2-29
- Item Hit Structure 2-31
- Help Item Structure 2-33

This document was created with FrameMaker 4.0.4

SmallFract Type 2-33
HSV Color Structure 2-34
HSL Color Structure 2-34
CMY Color Structure 2-35
Color Picker Manager Functions 2-36
Using the Standard Color Picker Dialog Box 2-36
`PickColor` 2-36
`GetColor` 2-37
Creating a Custom Color Picker Dialog Box 2-38
`CreateColorDialog` 2-38
`CreatePickerDialog` 2-39
`AddPickerToDialog` 2-40
`SetPickerVisibility` 2-41
`GetPickerVisibility` 2-42
`SetPickerPrompt` 2-42
`GetPickerOrigin` 2-43
`SetPickerOrigin` 2-44
`DisposeColorPicker` 2-44
Handling Events in a Custom Color Picker Dialog Box 2-45
`DoPickerEvent` 2-45
`DoPickerEdit` 2-46
`DoPickerDraw` 2-47
Getting Colors From and Setting Colors for a Custom Color Picker Dialog Box 2-47
`SetPickerColor` 2-48
`GetPickerColor` 2-49
Getting the Menu State and the Help Balloons for a Color Picker 2-50
`GetPickerEditMenuState` 2-50
`ExtractPickerHelpItem` 2-51
Setting and Getting Color-Matching Profiles for a Color Picker 2-52
`SetPickerProfile` 2-52
`GetPickerProfile` 2-53
Converting Colors Among Color Models 2-54
`CMY2RGB` 2-54
`RGB2CMY` 2-55
`HSL2RGB` 2-55
`RGB2HSL` 2-56
`HSV2RGB` 2-56

RGB2HSV 2-57
Converting Between SmallFract and Fixed Values 2-57
Fix2SmallFract 2-57
SmallFract2Fix 2-58
Application-Defined Functions 2-58
Handling Application-Directed Events in a Color Picker 2-58
MyPickerFilterFunction 2-59
Changing Colors in a Document 2-59
MyColorChangedFunction 2-60
Color Picker–Defined Functions 2-60
Setting Up a Color Picker 2-61
MyTestGraphicsWorld 2-62
MyInitPicker 2-62
MyGetDialog 2-64
MyGetItemList 2-64
MySetVisibility 2-65
Responding to Requests to Return and Set Color Picker Information 2-66
MyGetColor 2-66
MySetColor 2-67
MySetBaseItem 2-68
MyGetIconData 2-68
MyGetPrompt 2-69
MySetPrompt 2-70
MySetOrigin 2-71
MyGetProfile 2-72
MySetProfile 2-73
MyGetEditMenuState 2-74
MyExtractHelpItem 2-75
Responding to Events in a Color Picker 2-76
MyDrawPicker 2-76
MyDoEvent 2-77
MyItemHit 2-78
MyDoEdit 2-79
Result Codes 2-80

The section describes the constants, data structures, and functions defined for your application's use by the Color Picker Manager. This section also describes the functions your application or color picker may define for the Color Picker Manager to call.

Constants and Data Structures

The section describes the constants and data structures defined for your application's use by the Color Picker Manager.

Gestalt Selector for the Color Picker

To test for the availability and version of the Color Picker Manager, use the `Gestalt` function with the selector defined by the following enumerator:

```
enum {
    gestaltColorPickerVersion   = 'cpkr' /* returns version of 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.

Picker Actions

When your application uses the `DoPickerEvent` function (described on page 2-44) to pass an event to a color picker for handling, your application passes the event in an `EventData` structure (described on page 2-27). The color picker handling the event in turn uses the `action` field of the `EventData` structure to report the nature of the event. When your application uses the `DoPickerEdit` function (described on page 2-46), it passes information about the editing operation in an `EditData` structure (described on page 2-29). The color picker handling the event uses the `action` field of the `EditData` structure to report on the nature of the event.

The actions reported in this field are defined by the `PickerAction` enumeration.

This document was created with FrameMaker 4.0.4

```
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 */
    kApplItemHit            /* Dialog Manager returned an item in an
                               application-owned dialog box */
};
typedef short PickerAction;
```

Enumerator descriptions

`kDidNothing` The user performed no action worth reporting.

`kColorChanged` The user chose a different color. Your application may need to call the `GetPickerColor` function to obtain the chosen color; how your application handles this action for its own application-owned dialog boxes depends on your application. For dialog boxes owned by the system or the color picker, your application should probably wait until the user clicks the OK button before treating the color as final.

`kOkHit` The user clicked the OK button in a dialog box owned by the system or the color picker. Your application should save the newly chosen color.

`kCancelHit` The user clicked the Cancel button in a dialog box owned by the system or the color picker. Your application should restore the previously selected color and use the `DisposeColorPicker` function (described on page 2-44) to dispose of the color picker.

`kNewPickerChosen`

The user chose a new color picker from the More Choices list in a system-owned dialog box. Because this constant is returned only for system-owned dialog boxes, your application generally does not to respond to this action.

`kApplItemHit` The Dialog Manager returned an item number for an item in an application-owned dialog box. Your application must handle the event.

Color Types

A color picker maintains an original and a new color. When your application uses the `GetPickerColor` function (described on page 2-49), your application uses the `ColorTypes` enumeration to specify whether the color picker returns the original or the new color. When your application uses the `SetPickerColor` function (described on page 2-48), your application uses the `ColorTypes` enumeration to specify whether the color picker sets the original or the new color.

```
enum ColorTypes {
    kOriginalColor,         /* the original color */
    kNewColor               /* the new color chosen by the user */
};
typedef short ColorType;
```

Edit Menu Operations

If the user chooses an Edit command (or its keyboard equivalent) that applies to a color picker, your application can use the `DoPickerEdit` function (described on page 2-46) to request the color picker to perform the operation. In the `EditData` structure passed to `DoPickerEdit`, your application uses the `EditOperations` enumeration to specify the operation to perform.

```
enum EditOperations {
    kCut,       /* perform the Cut command */
    kCopy,      /* perform the Copy command */
    kPaste,     /* perform the Paste command */
    kClear,     /* perform the Clear command */
    kUndo       /* perform the Undo command */
};
typedef short EditOperation;
```

Item Hit Modifiers

A color picker must respond to user events directed at any of its items. As described for the color picker–defined function `MyItemHit` (described on page 2-78), a color picker should respond to the event represented by the `ItemHitModifiers` enumeration.

```
enum ItemHitModifiers {
    kMouseDown,         /* mouse down on item */
    kKeyDown,           /* key down 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 */
};
```

Dialog Placement Specifiers

In the `placeWhere` field of the color picker parameter block, your application specifies where to place the color picker dialog box. (The color picker parameter block is described on page 2-20.) Your application uses the `DialogPlacementSpecifiers` enumeration to specify the position of the color picker dialog box:

```
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 dialog box on the main screen */
};
typedef short DialogPlacementSpec;
```

Picker Flags

In the `flags` field of the color picker parameter block, the `SystemDialogInfo` structure, the `PickerDialogInfo` structure, or the `ApplicationDialogInfo` structure, your application specifies characteristics for the color picker dialog box. (These structures are described on page 2-20, page 2-24, page 2-25, and page 2-26, respectively.)

```
#define DialogIsMoveable           1
#define DialogIsModal              2
#define CanModifyPalette           4
#define CanAnimatePalette          8
#define AppIsColorSyncAware        16
```

Constant descriptions

`DialogIsMoveable`
: If your application sets the bit represented by this constant when creating a custom dialog box, then the color picker dialog box is moveable by the user.

`DialogIsModal`
: If your application sets the bit represented by this constant when creating a custom dialog box, then the color picker dialog box is a modal dialog box.

`CanModifyPalette`
: Your application should set the bit represented by this constant if your application can install a palette of its own that may modify (but not animate) the current color table. If you don't want the colors in the document to change as the user makes choices in the color picker dialog box, don't set this flag. See the chapter "Palette Manager" for more information about using color palettes.

`CanAnimatePalette`
: If your application sets the bit represented by this constant, then the color picker may modify or animate the palette.

`AppIsColorSyncAware`
: Your application should set the bit represented by this constant if your application uses ColorSync color matching. If your application sets this bit, a color may be returned to your application in a different color space than the one initially passed to the `PickColor` function. For example, your application could pass an RGB color with no color-matching profile in the field `theColor` in the color picker parameter block, and the Color Picker Manager could return a CMYK color with its associated profile. If your application does not set this flag, the Color Picker Manager automatically converts any color it receives back from the color picker to an RGB color.

Important

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" in *Advanced Color Imaging on the Mac OS* 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. ▲

The color picker may set any of the following flags and override your 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
                                            choices list */
```

Picker Attributes

In a resource of type `'thng'` that defines a color picker component, bits 23 to 0 in the `componentFlags` field specify attributes for the color picker. These bits can be represented by the following constants:

```
#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 CanDoApplDialog     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 color picker can be detached
                                    from a system-owned dialog box */
```

Event Forecasters

Your application can send event forecasters to warn the color picker about potential user actions. 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 the `forecast` field to an appropriate constant from the following list:

```
enum EventForcasters {
    kNoForcast,             /* no forcast (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 to an edit field */
```

```
    kNormalMouseDown        /* a normal click in the color picker's
                                focus */
};
typedef short EventForcaster;
```

For more information, see “Sending Event Forecasters to the Color Picker” in the chapter “Color Picker Manager” in *Advanced Color Imaging on the Mac OS*. The `EventData` structure is described on page 2-27, and the `DoPickerEvent` function is described on page 2-45.

Request Codes

When a color picker receives a request code from the Component Manager, the color picker determines the nature of the request, performs the appropriate processing, sets an error code if necessary, and returns an appropriate function result to the Component Manager. These request codes are defined by the `PickerMessages` enumeration.

```
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 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 color-
                                matching 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 information about local
                                coordinate system of dialog box */
    kExtractHelpItem        /* return information about help balloons */
} PickerMessages;
```

Enumerator descriptions

`kInitPicker`
: After receiving this request code, a color picker initializes any private data that it needs. See `MyInitPicker` (page 2-62) for more information about how a color picker should respond to this request code.

`kTestGraphicsWorld`
: After receiving this request code, a color picker determines whether it can operate on the user's system. See `MyTestGraphicsWorld` (page 2-62) for more information about how a color picker should respond to this request code.

`kGetDialog`
: After receiving this request code, a color picker returns `nil` if it uses the default dialog box, or it returns a pointer to its own dialog box. See `MyGetDialog` (page 2-64) for more information about how a color picker should respond to this request code.

`kGetItemList`
: After receiving this request code, a color picker returns a list of items for display in a color picker dialog box. See `MyGetItemList` (page 2-64) for more information about how a color picker should respond to this request code.

`kGetColor`
: After receiving this request code, a color picker returns a color—either the original color for the color picker or the new color selected by the user. See `MyGetColor` (page 2-66) for more information about how a color picker should respond to this request code.

`kSetColor`
: After receiving this request code, a color picker sets either the original color or the new color. See `MySetColor` (page 2-67) for more information about how a color picker should respond to this request code.

`kEvent` After receiving this request code, a color picker performs any special processing for an event. See `MyDoEvent` (page 2-77) for more information about how a color picker should respond to this request code.

`kEdit` After receiving this request code, a color picker performs an editing command or lets the Dialog Manager handle the command. See `MyDoEdit` (page 2-79) for more information about how a color picker should respond to this request code.

`kSetVisibility` After receiving this request code, a color picker changes its visibility. See `MySetVisibility` (page 2-65) for more information about how a color picker should respond to this request code.

`kDrawPicker` After receiving this request code, a color picker redraws itself. See `MyDrawPicker` (page 2-76) for more information about how a color picker should respond to this request code.

`kItemHit` After receiving this request code, a color picker responds to an event in a dialog box item. See `MyItemHit` (page 2-78) for more information about how a color picker should respond to this request code.

`kSetBaseItem` After receiving this request code, a color picker sets the base item for dialog box items. See `MySetBaseItem` (page 2-68) for more information about how a color picker should respond to this request code.

`kGetProfile` After receiving this request code, a color picker returns a handle to its destination color-matching profile. See `MyGetProfile` (page 2-72) for more information about how a color picker should respond to this request code.

`kSetProfile` After receiving this request code, a color picker changes its destination color-matching profile. See `MySetProfile` (page 2-73) for more information about how a color picker should respond to this request code.

`kGetPrompt` After receiving this request code, a color picker returns its prompt string. See `MyGetPrompt` (page 2-69) for more information about how a color picker should respond to this request code.

`kSetPrompt` After receiving this request code, a color picker sets its prompt string. See `MySetPrompt` (page 2-70) for more

information about how a color picker should respond to this request code.

`kGetIconData` After receiving this request code, a color picker returns its script code and the resource ID of its icon family. See `MyGetIconData` (page 2-68) for more information about how a color picker should respond to this request code.

`kGetEditMenuState` After receiving this request code, a color picker returns information about its edit menu. See `MyGetEditMenuState` (page 2-74) for more information about how a color picker should respond to this request code.

`kSetOrigin` After receiving this request code, a color picker updates any information it maintains about the local coordinate system of its dialog box. See `MySetOrigin` (page 2-71) for more information about how a color picker should respond to this request code.

`kExtractHelpItem` After receiving this request code, a color picker returns information about its help balloons. See `MyExtractHelpItem` (page 2-75) for more information about how a color picker should respond to this request code.

Picker Color Structure

For defining colors, version 2.0 of the Color Picker Manager uses a picker color structure. For example, when your application creates a color picker parameter block to pass to the `PickColor` function (described on page 2-36), your application supplies a picker color structure. The color that your application supplies in this field is passed to the color picker for editing. After the user clicks either the OK or Cancel button to close the dialog box, this field contains the color last chosen by the user.

The picker color structure includes a ColorSync 1.0 profile, a structure that matches colors among hardware devices such as displays, printers, and scanners. This color-matching profile (a data structure of type `CMProfile`) defines the color space of the color (which includes the type of color—RGB, CMYK, HSL, and so on). Using the `dstProfile` field of the color picker parameter block (described on page 2-20) or the `SetPickerProfile` function (described on page 2-52), your application can specify a destination color-matching profile, which describes the color space of the device for which

the color is being chosen. Given information about the destination profile, color pickers that are ColorSync aware can help the user choose a color that's within the destination device's gamut.

IMPORTANT

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" in *Advanced Color Imaging on the Mac OS* 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. ▲

The picker color structure is defined as follows:

```
typedef struct PMColor {
    CMProfileHandle     profile;
    CMColor             color;
} PMColor,*PMColorPtr;
```

Field descriptions

`profile` — A handle to a color-matching profile (`CMProfile` structure). If your application sets this field to `nil`, then the Color Picker Manager uses the default system profile.

`color` — A color, as specified in a color-matching (`CMColor`) structure, shown here.

```
typedef union {
 RGBColor   rgb;            /* an RGB color */
 XYZColor   xyz;            /* an XYZ color */
 CMYKColor  cmyk;           /* a CMYK color */
```

```
    unsigned short
                reserved[4];      /* reserved */
} CMColor, *CMColorList;
```

Picker Structure

When your application uses one of the Color Picker Manager's low-level calls, your application must specify the color picker by using the `picker` structure, which is defined as shown here.

```
typedef struct PrivatePickerRecord **picker;
```

Picker Icon Structure

A color picker responds to the `kGetIconData` request code by returning its script code and the resource ID of its icon family. The Color Picker Manager needs this information to display the color picker in the More Choices list. The color picker returns this information in a picker icon data (`PickerIconData`) structure.

```
typedef struct PickerIconData {
    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 */
} PickerIconData;
```

Field descriptions

`scriptCode` — The script code for the text used for the color picker's name. See *Inside Macintosh: Text* for more information about script codes.

`iconSuiteID` — The resource ID for the color picker's icon family. See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for information about icon families.

`helpResType` — The resource type for the help balloon used for the color picker's icon. See the chapter "Help Manager" for information about help balloons.

`helpResID` — The resource ID for the help balloon used for the color picker's icon.

See page 2-68 for more information about how a color picker should respond to the `kGetIconData` request code.

Picker Initialization Structure

A color picker responds to the `kInitPicker` request code by initializing any private data it needs.

```
typedef struct PickerInitData {
    DialogPtr   pickerDialog;   /* pointer to dialog box */
    DialogPtr   choicesDialog;  /* pointer to More Choices dialog box */
    long        flags;          /* color picker flags */
    picker      yourself;       /* the color picker */
} PickerInitData;
```

Field descriptions

`pickerDialog` — A pointer to the dialog box in which the color picker appears.

`choicesDialog` — A pointer to the More Choices dialog box.

`flags` — An integer in which any of the following flags may be set; see "Picker Flags" (page 2-8) for more information about these flags.

```
#define DialogIsMoveable        1
#define DialogIsModal           2
#define CanModifyPalette        4
#define CanAnimatePalette       8
#define AppIsColorSyncAware     16
#define InSystemDialog          32
#define InApplicationDialog     64
#define InPickerDialog          128
#define DetachedFromChoices     256
```

`yourself` — Your color picker.

Event Filter Function

When using the `PickColor` function, your application should set the `eventProc` field of the color picker parameter block to point to an event filter function that

handles events meant for your application. This function is defined by the Color Picker Manager as follows:

```
typedef pascal Boolean (*UserEventProc)(EventRecord *event);
```

See page 2-36 for information about the `PickColor` function. See page 2-20 for more information about the color picker parameter block. See page 2-59 for information about defining your own event filter function.

Color-Changed Function

When using the `PickColor` function, your application can set the `colorProc` field of the color picker parameter block to point to a function, described in detail on page 2-60, that updates colors in a document as the user selects them. This function is defined by the Color Picker Manager as follows:

```
typedef pascal void (*ColorChangedProc)(long userData,
                                        PMColorPtr newColor);
```

Edit Menu Items Structure

The `MenuItemInfo` structure is contained in the `PickerDialogInfo`, `SystemDialogInfo`, and `ApplicationDialogInfo` structures; it allows your application to specify your Edit menu for use when a color picker dialog box is displayed.

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

Descriptions for the `SystemDialogInfo`, `PickerDialogInfo`, and `ApplicationDialogInfo` structures begin on page 2-24.

Edit Menu State Structure

If your application is handling its own menus, and the color picker dialog box is the active window, your application needs to determine the color picker's specifications for the Edit menu. The `GetPickerEditMenuState` function (described on page 2-50) returns these specifications in a `MenuState` structure.

```
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;
```

Field descriptions

`cutEnabled` If the value returned in this field is `true`, then the Cut menu item is enabled; if the value is `false`, then the item is disabled.

`copyEnabled` If the value returned in this field is `true`, then the Copy menu item is enabled; if the value is `false`, then the item is disabled.

`pasteEnabled` If the value returned in this field is `true`, then the Paste menu item is enabled; if the value is `false`, then the item is disabled.

`clearEnabled` If the value returned in this field is `true`, then the Clear menu item is enabled; if the value is `false`, then the item is disabled.

`undoEnabled` If the value returned in this field is `true`, then the Undo menu item is enabled; if the value is `false`, then the item is disabled.

`undoString` The text for the Undo menu item.

Color Picker Parameter Block

When your application calls the `PickColor` function (described on page 2-36) to display a standard color picker dialog box, your application uses a color picker parameter block to specify information to and obtain information from the

Color Picker Manager. The color picker parameter block is defined by the `ColorPickerInfo` data type.

```
typedef struct ColorPickerInfo {    /* color picker parameter block */
    PMColor             theColor;       /* a picker color */
    CMProfileHandle     dstProfile;     /* destination profile */
    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-changed function */
    long                colorProcData;  /* data for color-changed
                                           function */
    Str255              prompt;         /* color picker prompt */
    MenuItemInfo        mInfo;          /* application's edit menu
                                           items */
    Boolean             newColorChosen; /* whether user changed color */
} ColorPickerInfo;
```

Field descriptions

`theColor` A picker color (`PMColor`) structure, described on page 2-15. The color that your application supplies in this field is passed to the color picker for editing. This becomes the ***original color*** for the color picker. After the user clicks either the OK or Cancel button to close the dialog box, this field contains the new color, that is, the color last chosen by the user. Although the new colors selected by the user may vary widely, the original color remains fixed for comparison. Figure 2-1 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* shows how the standard dialog box displays both the original and the new colors.

`dstProfile` A handle to a ColorSync 1.0 profile for the final output device. To use the default system profile, set this field to `nil`.

Important
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" in *Advanced Color Imaging on the Mac OS* 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. ▲

`flags` Bits representing the color picker flags, which are described in detail in "Picker Flags" (page 2-8). Your application can set any of the following flags:

```
#define CanModifyPalette              4
#define CanAnimatePalette             8
#define AppIsColorSyncAware           16
```

The color picker may set any of the following flags and override your application settings:

```
#define InSystemDialog                32
#define InApplicationDialog           64
#define InPickerDialog                128
#define DetachedFromChoices           256
```

`placeWhere` A specification for where to position the dialog box. Your application uses one of the following constants (described in detail on page 2-8) to specify the position for the color picker dialog box:

```
enum DialogPlacementSpecifiers {
    kAtSpecifiedOrigin,
    kDeepestColorScreen,
    kCenterOnMainScreen
};
typedef short DialogPlacementSpec;
```

`dialogOrigin` The point, in global coordinates, at which to locate the upper-left corner of the dialog box. This origin point is used only if your application supplies the `kAtSpecifiedOrigin` specifier in the `placeWhere` field.

`pickerType` The component subtype of the initial color picker. If your application sets this field to 0, the default color picker is used (that is, the last color picker chosen by the user). When `PickColor` returns, this field contains the component subtype of the color picker that was chosen when the user closed the color picker dialog box.

`eventProc` A pointer to an application-defined event 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. The event filter function you can supply here is described in detail on page 2-59.

`colorProc` A pointer to an application-defined function to handle color changes. This function, described in detail on page 2-60, should support the updating of colors in a document as the user selects them.

`colorProcData` A long integer that the Color Picker Manager passes to the application-defined function supplied in the `colorProc` field. Your application-defined function can use this value for any purpose it needs.

`prompt` A text string prompting the user to choose a color for a particular use (for example, "Choose a highlight color:").

`mInfo` A `menuItemInfo` structure, which is described on page 2-19.This structure allows your application to specify your Edit menu for use when a color picker dialog box is displayed

newColorChosen
: Upon completion, the `PickColor` functions the value `true` if the user chose a color and clicked the OK button, and `false` if the user clicked Cancel.

System-Owned Dialog Box Structure

The `SystemDialogInfo` structure contains the data required to create a system-owned dialog box for color pickers.

```
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;
```

Field descriptions

flags
: Bits representing the color picker flags, which are described in “Picker Flags” (page 2-8). Your application can set any of the following flags:

```
#define CanModifyPalette        4
#define CanAnimatePalette       8
#define AppIsColorSyncAware     16
```

The color picker may set any of the following flags and override your application settings:

```
#define InSystemDialog          32
#define InApplicationDialog     64
#define InPickerDialog          128
#define DetachedFromChoices     256
```

pickerType
: The component subtype of the color picker. If this field is set to 0, the default color picker is used (that is, the last color picker chosen by the user).

placeWhere
: A dialog placement constant, one of three values with which your application can specify whether the dialog box

should be centered on the deepest color screen, centered on the main screen, or placed at a specified location.

```
kAtSpecifiedOrigin      =   0;
kDeepestColorScreen     =   1;
kCenterOnMainSCreen     =   2;
```

`dialogOrigin` A point specifying placement of the upper-left corner of the dialog box, used if the `placeWhere` field contains the value represented by the `kAtSpecifiedOrigin` constant.

`mInfo` Information, stored in a `MenuItemInfo` structure, about the state of your application's Edit menu. The `MenuItemInfo` structure is described on page 2-19.

Picker-Owned Dialog Box Structure

The `PickerDialogInfo` structure contains the data required to create a color picker–owned color picker dialog box.

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

Field descriptions

`flags` Bits representing the color picker flags, which are described in “Picker Flags” (page 2-8). Your application can set any of the following flags:

```
#define CanModifyPalette            4
#define CanAnimatePalette           8
#define AppIsColorSyncAware         16
```

The color picker may set any of the following flags and override your application settings:

```
#define InSystemDialog              32
#define InApplicationDialog         64
#define InPickerDialog              128
#define DetachedFromChoices         256
```

`pickerType` The component subtype of the color picker. If this field is set to 0, the default color picker is used (that is, the last color picker chosen by the user).

`dialogOrigin` A pointer to the coordinates of the upper-left corner of the dialog box.

`mInfo` Information, stored in a `MenuItemInfo` structure, about the state of your application's Edit menu. The `MenuItemInfo` structure is described on page 2-19.

Application-Owned Dialog Box Structure

The `AddPickerToDialog` function, described on page 2-40, places a color picker into a dialog box. An application using the `AddPickerToDialog` function specifies a dialog box in a `ApplicationDialogInfo` structure.

```
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;
```

Field descriptions

`flags` Bits representing the color picker flags, which are described in detail on page 2-8. Your application can set any of the following flags:

```
#define CanModifyPalette        4
#define CanAnimatePalette       8
#define AppIsColorSyncAware     16
```

The color picker may set any of the following flags and override your application settings:

```
#define InSystemDialog           32
#define InApplicationDialog      64
#define InPickerDialog           128
#define DetachedFromChoices      256
```

`pickerType` The component subtype of the color picker. If this field is set to 0, the default color picker is used (that is, the last color picker chosen by the user).

`theDialog` A pointer to the dialog box to which to add the color picker specified to the `AddPickerToDialog` function.

`pickerOrigin` The coordinates of the upper-left corner of the color picker.

`mInfo` Information, stored in a `MenuItemInfo` structure, about the state of your application's Edit menu. The `MenuItemInfo` structure is described on page 2-19.

Event Data Structure

When your application uses the `DoPickerEvent` function to pass an event to a color picker for handling, your application uses an `EventData` structure to supply the color picker with information about the event, and to receive information about how the color picker handled the event. (The `DoPickerEvent` function is described on page 2-44.)

```
typedef struct EventData {
    EventRecord     *event;         /* an event record */
    PickerAction    action;         /* the action performed by
                                        the color picker */
    short           itemHit;        /* the item number for the item
                                        associated with the event */
    Boolean         handled;        /* true if the color picker
                                        handled the event */
    ColorChangedProc
                    colorProc;      /* application-defined function for
                                        changing colors in a document */
    long            colorProcData;  /* data used by application for
                                        function in ColorChangedProc
                                        field */
    EventForcaster  forcast;        /* event forecaster */
} EventData;
```

Field descriptions

`event` An event record. Your application supplies this field with an event to pass to the color picker. The event record is described in the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials.*

`action` The nature of the event. In this field, the Color Picker Manager returns a value, defined in the `PickerAction` enumeration, describing the event. See the description of the `PickerAction` enumeration on page 2-5 for a discussion about how your application should respond to these actions.

```
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;
```

`itemHit` For the item associated with the event, the number corresponding to the item's position with the item list resource of the color picker's dialog box. See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information about items, item list resources, and dialog boxes.

`handled` A Boolean value indicating whether the color picker or the Color Picker Manager handled the event. If the Color Picker Manager returns the value `true` in this field, then the event was handled; otherwise, your application should process the event.

If your application sends an event forecaster to the color picker, the color picker informs your application about whether the color picker is ready for the action to occur by setting this field to `true` if it's not ready and `false` if it is.

`colorProc` A pointer to an application-defined function to handle color changes. This function, described in detail on page 2-60, should support the updating of colors in a document as the user selects them.

`colorProcData` A long integer that the Color Picker Manager passes to the application-defined function supplied in the `colorProc` field. Your application-defined function can use this value for any purpose it needs.

`forcast` An event forecaster (that is, a warning) for the color picker. To send an event forecaster to the color picker, set the `event` field to `nil` and set the `forcast` field to a value from the following enumeration. (See "Sending Event Forecasters to the Color Picker" in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* for more information.)

```
enum EventForcasters {
 kNoForcast,        /* 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 the
                        color picker's focus */
};
typedef short EventForcaster;
```

Editing Data Structure

If the user chooses an Edit menu command for the color picker, your application needs to set the state of the Edit menu items according to the color

picker specifications and send the appropriate message to the color picker. Use the `GetPickerEditMenuState` function (described on page 2-50) to determine the state of the Edit menu items. Then use the `DoPickerEdit` function (described on page 2-46) to request the color picker to perform an editing operation. When your application uses the `DoPickerEdit` function, it passes information about the editing operation in an `EditData` structure; the color picker then uses this structure to describe whether and how it performed this operation.

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

Field descriptions

`theEdit` The editing operation to perform. Your application uses the `EditOperations` enumeration to specify the operation.

```
enum EditOperations {
    kCut,       /* perform the Cut command */
    kCopy,      /* perform the Copy command */
    kPaste,     /* perform the Paste command */
    kClear,     /* perform the Clear command */
    kUndo       /* perform the Undo command */
};
typedef short EditOperation;
```

`action` The nature of the event. In this field, the Color Picker Manager returns a value, defined in the `PickerAction` enumeration, describing the event. See the description of the `PickerAction` enumeration on page 2-5 for a discussion about how your application should respond to these actions.

```
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;
```

`handled` A Boolean value indicating whether the color picker or the Color Picker Manager handled the event. If the Color Picker Manager returns the value `true` in this field, then the event was handled; otherwise, your application should process the event.

Item Hit Structure

A color picker responds to the `kItemHit` request code by handling the event described in an `ItemHitData` structure. Your color picker also uses this structure to return information about any event handling it performs.

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

Field descriptions

`itemHit` The item receiving the event.

`ItemModifier` The action involving the item. These actions are represented by the following enumeration (which is described in greater detail on page 2-7):

```
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;
```

`action` The nature of the event. In this field, your color picker returns a value, defined in the `PickerAction` enumeration, describing the event. For more information about the `PickerAction` enumeration, see page 2-5.

```
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;
```

`colorProc` A pointer to an application-defined function to handle color changes. This function, described in detail on page 2-60, should support the updating of colors in a document as the user selects them. Your color picker should call this function.

`colorProcData` A long integer that your color picker passes to the application-defined function supplied in the `colorProc` field.

`where` Location, in coordinates local to the dialog box, of the mouse.

Help Item Structure

The `ExtractPickerHelpItem` function, described on page 2-51, reports the messages and other characteristics for the help balloons for a color picker. Help balloons are described in detail in the chapter "Help Manager" in *Inside Macintosh: More Macintosh Toolbox.*

```
typedef struct HelpItemInfo {   /* help balloon info */
    long            options;        /* 'hmnu' options bits */
    Point           tip;            /* tip location */
    Rect            altRect;        /* alternate rectangle */
    short           theProc;        /* res ID of balloon-definition
                                        function */
    short           variant;        /* variation code */
    HMMessageRecord helpMessage;    /* help message structure */
} HelpItemInfo;
```

Field descriptions

`options` Options set in the `'hmnu'` resource for the help balloon.

`tip` The location, in global coordinates, of the help balloon's tip.

`altRect` A rectangle, in global coordinates, that the Help Manager uses if necessary to calculate a new tip location for the help balloon.

`theProc` The resource ID of the `'WDEF'` resource containing the balloon-definition function for the help balloon.

`variant` The variation code (0–7) used to specify the shape and position of a help balloon.

`helpMessage` A help message (`HMMessageRecord`) structure, which describes the text for the help message of a help balloon.

SmallFract Type

The `SmallFract` type is derived from the low-order word of a fixed integer. The Color Picker Manager uses `SmallFract` values to save memory and to be compatible with the Color QuickDraw `RGBColor` structure. You can use the `Fix2SmallFract` function, described on page 2-57, to convert a fixed integer to a `SmallFract` value. You can use the `SmallFract2Fixed` function, described on page 2-58, to convert a `SmallFract` value to a fixed integer.

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

HSV Color Structure

The `HSVColor` structure contains the hue, saturation, and value of a color. Your application can use an `HSVColor` structure to specify a color in a `PMColor` structure. For example, your application supplies a `PMColor` structure in a color picker parameter block that it passes to the `PickColor` function.

```
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 most intensity */
};
typedef struct HSVColor HSVColor;
```

Field descriptions

`hue` The `SmallFract` value for the hue. (The `SmallFract` data type is described in the immediately preceding section.)

`saturation` The `SmallFract` value for the saturation, where 1 is full color.

`value` The `SmallFract` value for the color's value, where 1 is maximum intensity.

The `PMColor` structure is described on page 2-15, the `PickColor` function is described on page 2-36, and the color picker parameter block is described on page 2-20.

HSL Color Structure

The `HSLColor` structure contains a color's hue, saturation, and lightness values. Your application can use an `HSLColor` structure to specify a color in a `PMColor` structure. For example, your application supplies a `PMColor` structure in a color picker parameter block that it passes to the `PickColor` function. Note that the standard HLS order is altered to HSL.

```
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;
```

Field descriptions

`hue` — The `SmallFract` value for the hue. (The `SmallFract` data type is described on page 2-33.)

`saturation` — The `SmallFract` value for the saturation, where 1 is full color.

`lightness` — The `SmallFract` value for lightness, where 1 is full white.

The `PMColor` structure is described on page 2-15, the `PickColor` function is described on page 2-36, and the color picker parameter block is described on page 2-20.

CMY Color Structure

The `CMYColor` structure contains cyan, magenta, and yellow colors. Your application can use a `CMYColor` structure to specify a color in a `PMColor` structure. For example, your application supplies a `PMColor` structure in a color picker parameter block that it passes to the `PickColor` function. Note that CMY and RGB colors are complementary.

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

Field descriptions

`cyan` — The `SmallFract` value for the cyan component. (The `SmallFract` data type is described on page 2-33.)

`magenta` — The `SmallFract` value for the magenta component.

`yellow` — The `SmallFract` value for the yellow component.

The `PMColor` structure is described on page 2-15, the `PickColor` function is described on page 2-36, and the color picker parameter block is described on page 2-20.

Color Picker Manager Functions

The section describes the functions defined by the Color Picker Manager for your application's use.

Using the Standard Color Picker Dialog Box

Your application can use the `PickColor` function described in this section to present color pickers to the user and request the user to choose a color. The `GetColor` function, which is also described in this section, provides functionality similar to `PickColor`, except that `GetColor` does not support ColorSync 1.0 color-matching capabilities.

PickColor

To use the standard color picker dialog box, you should use the `PickColor` function.

```
pascal OSErr PickColor (ColorPickerInfo *theColorInfo);
```

`theColorInfo` A color picker parameter block, which is described on page 2-20.

DESCRIPTION

The `PickColor` function displays the standard, modal dialog box for color pickers. Use the color picker parameter block pointed to in the parameter `theColorInfo` to specify information to and obtain information from the Color Picker Manager.

When the user clicks the OK button, the `PickColor` function removes the dialog box and returns `true` in the `newColorChosen` field of the color picker parameter block pointed to by the `theColorInfo` parameter. The `PickColor` function also returns the user's selected color in the field `theColor`. When the user clicks the

Cancel button, `PickColor` removes the dialog box and returns `false` in the `newColorChosen` field.

SEE ALSO

Listing 2-1 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to use the `PickColor` function. If your application needs to display color pickers in a dialog box other than the standard, modal dialog box, your application must use the Color Picker Manager low-level functions, as described in "Using Customized Dialog Boxes for Color Pickers" in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS*.

GetColor

To display a standard Color Picker dialog box, you should use the `PickColor` function, which is described in the immediately preceding section. The `GetColor` function, which is described here, was designed for use for version 1.0 of the Color Picker Package and is still supported for backward compatibility.

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

`where` A point defining the location of the upper-left corner of the dialog box. If you set this parameter to (0,0), the dialog box is centered horizontally on the main screen, with one-third of the empty space above the box and two-thirds below, regardless of the screen size. If you set this parameter to (–1,–1), the `GetColor` function displays the dialog box on the screen supporting the greatest pixel depth.

`prompt` Text for prompting the user to choose a color. This string is displayed in the upper-left corner of the dialog box.

`inColor` An `RGBColor` structure for a color at entry to the picker. This is the original color, which the user may want for comparison.

`outColor` An `RGBColor` structure describing the new color. This is set to the last color that the user picked before clicking OK. On entry, the `outColor` parameter is treated as undefined, so the output color sample initially matches the input. Although the color being picked may vary widely, the input color sample remains fixed, and clicking the input sample resets the output color sample to match it.

DESCRIPTION

The `GetColor` function displays a standard color picker dialog box onscreen. The `GetColor` function returns `true` and removes the dialog box when the user clicks the OK button; `GetColor` returns `false` and removes the dialog box when the user clicks the Cancel button.

SPECIAL CONSIDERATIONS

The `GetColor` function does not support ColorSync 1.0 color matching; however, the `PickColor` function does.

Creating a Custom Color Picker Dialog Box

If your application needs to create a dialog box other than the standard, modal dialog box used to display color pickers, your application can use the Color Picker Manager low-level functions described in this section.

CreateColorDialog

To create a system-owned color picker dialog box, use the `CreateColorDialog` function.

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

`info` A `SystemDialogInfo` structure, which is described on page 2-24.

thePicker — The last color picker chosen by the user. Your application often refers to the color picker returned here in subsequent Color Picker Manager functions.

DESCRIPTION

The `CreateColorDialog` function creates a system-owned color picker dialog box of the type requested in the `info` parameter and places inside it the color picker returned in the parameter `thePicker`. The dialog box is invisible upon creation; use the `SetPickerVisibility` function (described on page 2-41) to make it visible. The Color Picker Manager may change some of the flags in the `SystemDialogInfo` structure (such as those regarding the type of dialog box for the color picker).

SEE ALSO

Listing 2-4 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to use the `CreateColorDialog` function.

CreatePickerDialog

To create a color picker–owned dialog box, use the `CreatePickerDialog` function.

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

info — A `PickerDialogInfo` structure, which is described on page 2-25.

thePicker — The last color picker chosen by the user. Your application often refers to the color picker returned here in subsequent Color Picker Manager functions.

DESCRIPTION

The `CreatePickerDialog` function creates a color picker–owned color picker dialog box of the type requested in the `info` parameter and places inside it the color picker returned in the parameter `thePicker`.

If the color picker does not have a private dialog box, the Color Picker Manager creates a modeless dialog box by default. The dialog box is invisible upon creation; use the `SetPickerVisibility` function (described on page 2-41) to make it visible. The Color Picker Manager may change some of the flags in the `PickerDialogInfo` structure (such as those regarding the type of dialog box for the color picker).

SEE ALSO

Listing 2-6 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to use the `CreatePickerDialog` function.

AddPickerToDialog

To add a color picker to an application-owned dialog box, use the `AddPickerToDialog` function.

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

`info` A `ApplicationDialogInfo` structure, which is described on page 2-26.

`thePicker` The last color picker chosen by the user. Your application often refers to the color picker returned here in subsequent Color Picker Manager functions.

DESCRIPTION

The `AddPickerToDialog` function places the color picker returned in the parameter `thePicker` inside the dialog box specified by the `info` parameter. All of your application's dialog items must already appear in the dialog box, because your application may not add dialog items after the dialog box has been created. The dialog box is invisible upon creation; use the `SetPickerVisibility` function (described next) to make it visible. The Color Picker Manager may change some of the flags in the `ApplicationDialogInfo` structure (such as those regarding the type of dialog box for the color picker).

SEE ALSO

Listing 2-5 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to use the `AddPickerToDialog` function.

SetPickerVisibility

To make a custom dialog box for a color picker visible or invisible, use the `SetPickerVisibility` function.

```
pascal OSErr SetPickerVisibility (
                    picker thePicker,
                    short visible);
```

`thePicker` A color picker.

`visible` The visibility state of the color picker.

DESCRIPTION

The `SetPickerVisibility` function sets the visibility state of the dialog box containing the color picker specified in the parameter `thePicker`.

SEE ALSO

Listing 2-4 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to use the `SetPickerVisibility` function. To determine whether a color picker is visible or not, you can use the `GetPickerVisibility` function, which is described next.

GetPickerVisibility

To determine whether a custom dialog box for a color picker is visible or invisible, use the `GetPickerVisibility` function.

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

`thePicker` A color picker.

`visible` The visibility state of the color picker dialog box. If the value pointed to in this parameter is `true` when `GetPickerVisibility` completes, then the dialog box is visible; if the value is `false`, then it's invisible.

DESCRIPTION

In the `vis` parameter, the `GetPickerVisibility` function returns a Boolean value indicating whether the a custom dialog box for the color picker specified in the parameter `thePicker` is visible (`true`) or invisible (`false`).

SEE ALSO

To change the visibility, use the `SetPickerVisibility` function, which is described in the preceding section.

SetPickerPrompt

To set the prompt for a custom dialog box for a color picker, you can use the `SetPickerPrompt` function.

```
pascal OSErr SetPickerPrompt (
                    picker thePicker,
                    Str255 promptString);
```

`thePicker` A color picker.

`promptString` A text string prompting the user to choose a color for a particular use (for example, "Choose a highlight color:").

DESCRIPTION

The `SetPickerPrompt` function sets the dialog box for the color picker specified in the parameter `thePicker` to display the prompt supplied in the `promptString` parameter.

To set the prompt for a standard, modal dialog box, use the `prompt` field of the color picker parameter block, which is described on page 2-20.

SEE ALSO

Listing 2-4 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to use the `SetPickerPrompt` function.

GetPickerOrigin

To determine the location of the upper-left corner of a dialog box containing a color picker, use the `GetPickerOrigin` function.

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

`thePicker` A color picker.

`where` The global coordinates for the upper-left corner of the dialog box.

DESCRIPTION

In the value pointed to by the `where` parameter, the `GetPickerOrigin` function returns the global coordinates of the origin of the dialog box containing the color picker specified in the parameter `thePicker`.

SetPickerOrigin

To move an application-owned dialog box for a color picker, use the `SetPickerOrigin` function.

```
pascal OSErr SetPickerOrigin (
                        picker thePicker,
                        Point where);
```

`thePicker` A color picker.

`where` The global coordinates of where to move the upper-left corner of the dialog box containing the color picker.

DESCRIPTION

The `SetPickerOrigin` function moves the upper-left corner of dialog box containing the color picker specified in the parameter `thePicker` to the location specified in the `where` parameter.

SPECIAL CONSIDERATIONS

The `SetPickerOrigin` function works only for application-owned dialog boxes; system-owned and color picker–owned dialog boxes cause `SetPickerOrigin` to return an error.

DisposeColorPicker

To dispose of a color picker, use the `DisposeColorPicker` function.

```
pascal OSErr DisposeColorPicker (picker thePicker);
```

`thePicker` A color picker.

DESCRIPTION

The `DisposeColorPicker` function disposes of the memory allocated for the color picker specified in the parameter `thePicker`. This function also disposes of the memory for its dialog box and dialog items.

SEE ALSO

Listing 2-9 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to use the `DisposeColorPicker` function.

Handling Events in a Custom Color Picker Dialog Box

This section describes the functions that your application can use to handle events when displaying a custom dialog box for a color picker.

DoPickerEvent

To pass an event to a color picker for handling, use the `DoPickerEvent` function.

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

`thePicker` The color picker to handle the event.

`data` An `EventData` structure, which is described on page 2-27. You use this structure to supply a color picker with information about an event, and to receive information about how the color picker handled the event.

DESCRIPTION

The `DoPickerEvent` function uses the `EventData` structure you point to in the `data` parameter to pass an event to the color picker specified in the parameter `thePicker`. If the color picker handles the event, it returns the value `true` in the `handled` field of the `EventData` structure; otherwise, it returns the value `false`, in which case your application should continue handling the event.

SPECIAL CONSIDERATIONS

The `DoPickerEvent` function calls the Dialog Manager function `DialogSelect`. If your application needs to filter or preprocess events before `DialogSelect` handles them, your application must do so before calling `DoPickerEvent`.

SEE ALSO

Listing 2-9 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to use the `DoPickerEvent` function. Event handling on the Macintosh is described in the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials.* The Dialog Manager and the `DialogSelect` function are described in the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials.*

DoPickerEdit

To request the color picker to perform an editing operation, use the `DoPickerEdit` function.

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

`thePicker` The color picker to perform the editing operation.

`data` An `EditData` structure described the editing operation to perform. (See page 2-29 for information about the `EditData` structure.)

DESCRIPTION

The `DoPickerEdit` function requests the color picker specified in the parameter `thePicker` to perform the editing operation specified in the `EditData` structure pointed to in the `data` parameter.

If the user chooses an Edit menu command for the color picker, your application needs to set the state of the Edit menu items according to the color picker specifications and send the appropriate message to the color picker. Use

the `GetPickerEditMenuState` function (described on page 2-50) to determine the state of the Edit menu items before calling `DoPickerEdit`.

SEE ALSO

Listing 2-10 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to use the `DoPickerEdit` function.

DoPickerDraw

To request a color picker to redraw itself (as, for example, in response to an update event), use the `DoPickerDraw` function.

```
pascal OSErr DoPickerDraw (picker thePicker);
```

`thePicker` The color picker to redraw itself.

DESCRIPTION

The `DoPickerDraw` function requests the color picker specified in the parameter `thePicker` to redraw itself.

Getting Colors From and Setting Colors for a Custom Color Picker Dialog Box

When creating a custom dialog box for color pickers, 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. You use the `SetPickerColor` function to set both colors.

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 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS*.

SetPickerColor

To set either the original or new color for a color picker, use the `SetPickerColor` function.

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

`thePicker` The color picker for which to set a color.

`whichColor` Either of two values: `kOriginalColor` or `kNewColor`.

`color` A pointer to a `PMColor` structure, which is described on page 2-15.

DESCRIPTION

The `SetPickerColor` function sets the color picker specified by the parameter `thePicker` to use the color specified in the `color` parameter. If your application passes `kOriginalColor` in the `whichColor` parameter, then `SetPickerColor` sets this color as the original color to be edited. If your application passes `kNewColor`, then `SetPickerColor` sets the color to be used as if it were the last color selected by the user.

Use the `SetPickerColor` function for setting colors for color pickers in custom dialog boxes. 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 new color until the user begins editing the color.

SEE ALSO

Listing 2-7 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to use the `SetPickerColor` function.

GetPickerColor

To obtain either the original or the new color from a color picker, use the `GetPickerColor` function.

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

`thePicker` The color picker from which to obtain a color.

`whichColor` Either of two values: `kOriginalColor` or `kNewColor`.

`color` A pointer to a `PMColor` structure, which is described on page 2-15.

DESCRIPTION

In the `PMColor` structure pointed to by the `color` parameter, the `GetPickerColor` function returns a color from the color picker specified by the parameter `thePicker`. If your application passes `kOriginalColor` in the `whichColor` parameter, then `GetPickerColor` returns the color that the user began editing. If your application passes `kNewColor`, then `GetPickerColor` returns the new color selected by the user.

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.

SEE ALSO

Listing 2-8 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to use the `GetPickerColor` function.

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

The functions described in this section allow your application to determine information about a color picker's Edit menu state, Edit menu items, and help balloon.

GetPickerEditMenuState

To determine a color picker's specifications for the Edit menu, use the `GetPickerEditMenuState` function.

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

`thePicker` The color picker whose Edit menu specifications you need to determine.

`mState` A pointer to a `MenuState` structure, which is described on page 2-20.

DESCRIPTION

In the `MenuState` structure pointed to in the `mState` parameter, the `GetPickerEditMenuState` function returns the state of the Edit menu that is needed by the color picker specified in the parameter `thePicker`.

Your application needs to make this call only if your application is handling its own menus and the color picker dialog box is the active window.

SEE ALSO

Listing 2-10 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to use the `GetPickerEditMenuState` function.

ExtractPickerHelpItem

The Color Picker Manager provides help balloons for color picker dialog boxes. For all types of color picker dialog boxes, applications generally don't need to determine or change the default help balloons. However, if your application absolutely requires greater control over Balloon Help, it can use the `ExtractPickerHelpItem` function to obtain and change the messages and other characteristics of the help balloons for a color picker.

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

`thePicker` The color picker whose help balloons you wish to obtain.

`itemNo` A number corresponding to the position of an item in the item list resource of the color picker's dialog box.

`whichState` For menu items and items in alert or dialog boxes, the state of the item specified in the `itemNo` parameter. The following constants are used to represent the possible states: `kHMEnabledItem`, `kHMDisabledItem`, `kHMCheckedItem`, and `kHMOtherItem`. See the chapter "Help Manager" in *Inside Macintosh: More Macintosh Toolbox* for descriptions of these states for various types of dialog items.

`helpInfo` A `HelpItemInfo` structure, as described on page 2-33. In the `helpMessage` field of this structure, the Color Picker Manager passes the default help message in an `HMHelpMessage` structure. The Color Picker Manager passes the default characteristics of the help balloon—that is, the value of the `options` element of its help resource, its tip location, its alternate rectangle, its tip function, and its variation code—in the rest of the fields of the `HelpItemInfo` structure.

DESCRIPTION

For the color picker specified in the parameter `thePicker`, the `ExtractPickerHelpItem` function reports the messages and other characteristics for its help balloons. If your application needs to override the help message or

another help balloon characteristic for the item specified in the `itemNo` parameter, your application should specify the desired help message and characteristics in the `HelpItemInfo` structure pointed to in the `helpInfo` parameter, and then use the Help Manager function `HMShowBalloon` to display the altered help balloon.

SEE ALSO

Listing 2-14 in the chapter “Color Picker Manager” in *Advanced Color Imaging on the Mac OS* illustrates how to use this function. See the chapter “Help Manager” in *Inside Macintosh: More Macintosh Toolbox* for more information about help balloons.

Setting and Getting Color-Matching Profiles for a Color Picker

The functions described in this section allow your application to set and obtain the destination profiles used by a color picker.

SetPickerProfile

To set the destination color-matching profile for a color picker, use the `SetPickerProfile` function.

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

`thePicker` A color picker.

`profile` A handle to a ColorSync 1.0 profile for the final output device.

DESCRIPTION

For the color picker specified in the parameter `thePicker`, the `SetPickerProfile` function sets the color-matching profile specified in the `profile` parameter to be the destination profile.

SPECIAL CONSIDERATIONS

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" in *Advanced Color Imaging on the Mac OS* 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.

SEE ALSO

Listing 2-12 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to use the `SetPickerProfile` function. To determine the destination color-matching profile currently used by a color picker, use the `GetPickerProfile` function, which is described next.

GetPickerProfile

To determine the destination color-matching profile currently used by a color picker, use the `GetPickerProfile` function.

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

`thePicker` A color picker.

`profile` A handle to a ColorSync 1.0 profile for the final output device.

DESCRIPTION

In the color-matching profile pointed to in the `profile` parameter, the `GetPickerProfile` function returns the destination profile currently used by the color picker specified in the parameter `thePicker`.

SPECIAL CONSIDERATIONS

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" in *Advanced Color Imaging on the Mac OS* 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.

SEE ALSO

Listing 2-13 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to use the `GetPickerProfile` function. To change the destination color-matching profile for a color picker, use the `SetPickerProfile` function, which is described in the preceding section.

Converting Colors Among Color Models

These functions convert between RGB colors and CMY, HLS, and HSV colors.

CMY2RGB

To convert a CMY color to its equivalent RGB color, use the `CMY2RGB` function.

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

`cColor` A `CMYColor` structure to be converted.

`rColor` An `RGBColor` structure for the converted color.

DESCRIPTION

The `CMY2RGB` function converts the CMY color specified in the `cColor` parameter to the RGB color pointed to in the `rColor` parameter.

RGB2CMY

To convert an RGB color to a CMY color, use the `RGB2CMY` function.

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

`rColor` An `RGBColor` structure to be converted.

`cColor` A `CMYColor` structure for the converted color.

DESCRIPTION

The `RGB2CMY` function converts the RGB color specified in the `rColor` parameter to the CMY color pointed to in the `cColor` parameter.

HSL2RGB

To convert an HSL color to an RGB color, use the `HSL2RGB` function.

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

`hColor` The `HSLColor` structure to be converted.

`rColor` An `RGBColor` structure for the converted color.

DESCRIPTION

The `HSL2RGB` function converts the HSL color specified in the `hColor` parameter to the RGB color pointed to in the `rColor` parameter.

RGB2HSL

To convert an RGB color to an HSL color, use the `RGB2HSL` function.

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

`rColor` The `RGBColor` structure to be converted.

`hColor` An `HSLColor` structure for the converted color.

DESCRIPTION

The `RGB2HSL` function converts the RGB color specified in the `rColor` parameter to the HSL color pointed to in the `hColor` parameter.

HSV2RGB

To convert an HSV color to an RGB color, use the `HSV2RGB` function.

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

`hColor` The `HSVColor` structure to be converted.

`rColor` An `RGBColor` structure for the converted color.

DESCRIPTION

The `HSV2RGB` function converts the HSV color specified in the `hColor` parameter to the RGB color pointed to in the `rColor` parameter.

RGB2HSV

To convert an RGB color to an HSV color, use the `RGB2HSV` function.

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

`rColor` The `RGBColor` structure to be converted.

`hColor` An `HSVColor` structure for the converted color.

DESCRIPTION

The `RGB2HSV` function converts the RGB color specified in the `rColor` parameter to the HSV color pointed to in the `hColor` parameter.

Converting Between SmallFract and Fixed Values

A `SmallFract` value can represent a value between 0 and 65,535. Introduced by the original Color Picker Package, `SmallFract` values are used in `CMYColor`, `HSLColor`, and `HSVColor` structures. You can use these functions if you need to convert `SmallFract` values to or from fixed values. (They can be assigned directly to and from integers.)

Fix2SmallFract

To convert a fixed integer to a `SmallFract` value, use the `Fix2SmallFract` function.

```
pascal SmallFract Fix2SmallFract (Fixed f);
```

`f` The value of type `Fixed` to be converted to a `SmallFract` value.

DESCRIPTION

The `Fix2SmallFract` function converts the fixed integer specified in the `f` parameter into a value of type `SmallFract` and returns this value as its result.

SmallFract2Fix

To convert a `SmallFract` value to a fixed integer, use the `SmallFract2Fix` function.

```
pascal Fixed SmallFract2Fix (SmallFract s);
```

`s` The value of type `SmallFract` value to be converted into a fixed integer.

DESCRIPTION

The `SmallFract2Fix` function converts the `SmallFract` value specified in the `s` parameter into a fixed integer, which is returned as the function result.

Application-Defined Functions

This section describes functions that your application can define for use by the Color Picker Manager. For example, the Color Picker Manager calls one function—defined in this section as `MyPickerFilterFunction`—when your application needs to handle events before the Color Picker Manager does. The Color Picker Manager calls the function defined in this section as `MyColorChangedFunction` when your application needs to update colors in a document as the user selects colors from a color picker.

Handling Application-Directed Events in a Color Picker

Applications can generally allow the Color Picker Manager to handle all events that might occur while displaying the standard dialog box. 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.

MyPickerFilterFunction

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. Your filter function should take one parameter—a pointer to an event record—and it should return a Boolean value. For example, this is how you would declare it if your were to name it `MyPickerFilterFunction`.

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

`event` An event record, which is described in the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials.*

DESCRIPTION

Your filter function should examine the event record passed in the first parameter to determine whether your application needs to handle the event contained in the record. If your application handles the event, your filter function should return `true` so that the Color Picker Manager won't process the event any further. If your application does not handle the event, your filter function should returns `false` so that the Color Picker Manager or the color picker can handle the event.

SEE ALSO

Listing 2-2 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* provides an example of an event filter function for a color picker.

Changing Colors in a Document

Your application can define a function that updates colors in a document as the user selects them from a color picker.

MyColorChangedFunction

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. Your function should take two parameters—one for data previously specified by your application, and another specifying the new color selected by the user. For example, this is how you would declare it if you were to name it `MyColorChangedFunction`.

```
pascal void MyColorChangedFunction (
                    long userData,
                    PMColorPtr newColor);
```

`userData` Data that your application supplies in the `colorProcData` field of the color picker parameter block (which is described on page 2-20). Your application can use this value for any purpose it needs.

`newColor` A pointer to a `PMColor` structure that contains the new color selected by the user. (The `PMColor` structure is described on page 2-15.)

DESCRIPTION

Your color-changed function should update the user's document to use the color specified in the `newColor` parameter.

SEE ALSO

Listing 2-3 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* provides an example of a color-changed function.

Color Picker–Defined Functions

If you are creating your own color picker, it must support the functions described in this section. When you create a color picker, the Color Picker Manager uses the Component Manager to request services from your color picker. The code for your color picker should be contained in a resource, as described in "Creating a Component Resource for a Color Picker" in the

chapter “Color Picker Manager” in *Advanced Color Imaging on the Mac OS*. The Component Manager expects that the entry point in this resource is a function having this format:

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

Listing 2-16 in the chapter “Color Picker Manager” in *Advanced Color Imaging on the Mac OS* illustrates how to implement this function.

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.

The request codes that the Component Manager sends to your color picker are described on page 2-12. Your color picker must also support the four required request codes described in the chapter “Component Manager” in *Inside Macintosh: More Macintosh Toolbox.*

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.

To extract these parameters, 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. Therefore, all of the functions described in this section take a handle to storage as their first parameter.

Setting Up a Color Picker

This section describes the functions that your color picker should define for setting up your color picker.

MyTestGraphicsWorld

If you create a color picker, it must respond to the `kTestGraphicsWorld` request code. So that your color picker can test whether it can operate under existing conditions, the Color Picker Manager gives your color picker a copy of the picker flags as if the Color Picker Manager were requesting the color picker to open. A color picker typically responds to the `kTestGraphicsWorld` request code by calling a color picker–defined subroutine (for example, `MyTestGraphicsWorld`) to handle the request.

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

`storage` A handle to your color picker's global data.

`data` A pointer to a `PickerInitData` structure, in which one or more color picker flags may be set by the application. The `PickerInitData` structure is described on page 2-18. Your color picker may need to change some of these flags (such as those indicating the type of dialog box in which it appears).

DESCRIPTION

Your `MyTestGraphicsWorld` function should return `noErr` if your color picker can operate on the current system with the restrictions pointed to in the `data` parameter. If your color picker cannot operate under these conditions, it should return a result code other than `noErr`.

SEE ALSO

Listing 2-18 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to implement this function.

MyInitPicker

If you create a color picker, it must respond to the `kInitPicker` request code. The Color Picker Manager sends this code to request your color picker to

instantiate any private data it needs. A color picker responds to the `kInitPicker` request code by calling a color picker–defined subroutine (for example, `MyInitPicker`) to handle the request.

```
pascal ComponentResult MyInitPicker (
                        PickerStorageHndl storage,
                        PickerInitData *data);
```

`storage` A handle to your color picker's newly initialized global data.

`data` A pointer to a `PickerInitData` structure, in which one or more Color Picker flags may be set. You may want your color picker to store this information in its global data. The `PickerInitData` structure is described on page 2-18.

DESCRIPTION

Using the storage allocated in the `storage` parameter, your `MyInitPicker` function should initialize any private data needed by your color picker.

The Color Picker Manager uses the Component Manager to send the `kInitPicker` request code after your color picker has set up all of its external data. If the Color Picker Manager has opened your color picker only to obtain a list of color pickers for the More Choices list, your color picker will not receive this message unless it is actually chosen by the user.

Before handling the `kInitPicker` request code, your color picker must be able to handle the `kTestGraphicsWorld`, `kGetDialog`, and `kGetItemList` request codes (described on page 2-62, page 2-64, and page 2-64, respectively).

Your function should return `noErr` if successful, or an appropriate result code otherwise.

SEE ALSO

Listing 2-17 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to implement this function.

MyGetDialog

If you create a color picker, it must respond to the `kGetDialog` request code. The Color Picker Manager sends this code to obtain the custom dialog box for your color picker, in case your color picker uses a custom dialog box. A color picker responds to the `kGetDialog` request code by calling a color picker–defined subroutine (for example, `MyGetDialog`) to handle the request.

```
pascal DialogPtr MyGetDialog (PickerStorageHndl storage);
```

`storage` A handle to your color picker's global data.

DESCRIPTION

If your color picker uses its own dialog box, your `MyGetDialog` function should return a pointer to this dialog box as its function result. If your color picker does not use a color picker–owned dialog box, your `MyGetDialog` function should return `nil`.

MyGetItemList

If you create a color picker, it must respond to the `kGetItemList` request code. The Color Picker Manager sends this request code to obtain the dialog items for your color picker. A color picker responds to the `kGetItemList` request code by calling a color picker–defined subroutine (for example, `MyGetItemList`) to handle the request.

```
pascal ComponentResult MyGetItemList (PickerStorageHndl storage);
```

`storage` A handle to your color picker's global data.

DESCRIPTION

Your `MyGetItemList` function should coerce a handle for one or more dialog items into a long integer and return this as a function result. The Color Picker Manager adds these items to the color picker dialog box. If your color picker has no items to add, it should return `nil`.

If your color picker saved the items in a dialog item list ('DITL') resource, your color picker should use the Resource Manager function GetResource to obtain the handle, and the Resource Manager function DetachResource to detach the resource.

SEE ALSO

Listing 2-19 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to implement this function.

MySetVisibility

If you create a color picker, it must respond to the kSetVisibility request code. The Color Picker Manager sends this code to request your color picker to hide or show itself. Your color picker should respond to the kSetVisibility request code by calling a color picker–defined subroutine (for example, MySetVisibility) to handle the request.

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

storage A handle to your color picker's global data.

visible A Boolean value, where true means your color picker should make itself visible and false means invisible.

DESCRIPTION

When passed true in the visible parameter, your MySetVisibility function should make your color picker visible; when passed false, your function should make your color picker invisible.

Your function should return noErr if successful, or an appropriate result code otherwise.

Responding to Requests to Return and Set Color Picker Information

This section describes the functions that your color picker should define for returning and setting information used by your color picker.

MyGetColor

If you create a color picker, it must respond to the `kGetColor` request code. The Color Picker Manager sends this code to request your color picker to return an original or a new color. A color picker responds to the `kGetColor` request code by calling a color picker–defined subroutine (for example, `MyGetColor`) to handle the request.

```
pascal ComponentResult MyGetColor (
                      PickerStorageHndl storage,
                      ColorType whichColor,
                      PMColorPtr color);
```

`storage` A handle to your color picker's global data.

`whichColor` A type of color—either original or new—requested from your color picker. Your function should respond to the value represented by either the `kOriginalColor` or `kNewColor` constant.

`color` A pointer to a `PMColor` structure, which is described on page 2-15.

DESCRIPTION

In the `PMColor` structure pointed to by the `color` parameter, your `MyGetColor` function should return a color. If your `MyGetColor` function is passed the value represented by the `kOriginalColor` constant, it should return the color that the user first begins to edit. If your `MyGetColor` function is passed the value represented by the `kNewColor` constant, it should return the last color selected by the user.

Your function should return `noErr` if successful, or an appropriate result code otherwise.

SEE ALSO

Listing 2-24 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to implement this function.

MySetColor

If you create a color picker, it must respond to the `kSetColor` request code. The Color Picker Manager sends this code to request your color picker to set either the original or the new color. A color picker responds to the `kSetColor` request code by calling a color picker–defined subroutine (for example, `MySetColor`) to handle the request.

```
pascal ComponentResult MySetColor (
                        PickerStorageHndl storage,
                        ColorType whichColor,
                        PMColorPtr color);
```

`storage` A handle to your color picker's global data.

`whichColor` A type of color—either original or new—which your color picker should set. Your function should respond to the value represented by either the `kOriginalColor` or `kNewColor` constant.

`color` A pointer to a `PMColor` structure, which is described on page 2-15.

DESCRIPTION

Your `MySetColor` function should set an original or a new color to that specified in the `color` parameter. If your `MySetColor` function is passed the value represented by the `kOriginalColor` constant, it should set the color that the user begins to edit. If your `MyGetColor` function is passed the value represented by the `kNewColor` constant, it should set the color to be used as if it were the last color selected by the user.

Your function should return `noErr` if successful, or an appropriate result code otherwise.

SEE ALSO

Listing 2-25 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to implement this function.

MySetBaseItem

If you create a color picker, it must respond to the `kSetBaseItem` request code. The Color Picker Manager sends this code to obtain the first item in your color picker's item list. A color picker responds to the `kSetBaseItem` request code by calling a color picker–defined subroutine (for example, `MySetBaseItem`) to handle the request.

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

`storage` A handle to your color picker's global data.

`baseItem` In the dialog item list, the number of the first item that belongs to your color picker.

DESCRIPTION

Your `MySetBaseItem` function allows your color picker to access its dialog items through the Dialog Manager, where

`RealItemNumber = baseItem + localItemNumber` (1 based)

Your function should return `noErr` if successful, or an appropriate result code otherwise.

MyGetIconData

If you create a color picker, it must respond to the `kGetIconData` request code. The Color Picker Manager sends this request code to obtain information about your color picker's icon family and script code. A color picker typically

responds to the `kGetIconData` request code by calling a color picker–defined subroutine (for example, `MyGetIconData`) to handle the request.

```
pascal ComponentResult MyGetIconData (
                        PickerStorageHndl storage,
                        PickerIconData *data);
```

`storage` A handle to your color picker's global data.

`data` A pointer to a `PickerIconData` structure returned by your color picker. (The `PickerIconData` structure is described on page 2-17.) In the `scriptCode` field of this structure, your color picker should return its script code, and in the `iconSuiteID` field, your color picker should return the resource ID of its icon family

DESCRIPTION

Your `MyGetIconData` function should return the data that the Color Picker Manager needs to display your picker in the More Choices list—specifically the script code for its name and the resource ID of your color picker's icon family. See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for information about icon families.

Your function should return `noErr` if successful, or an appropriate result code otherwise.

SEE ALSO

Listing 2-26 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to implement this function.

MyGetPrompt

If you create a color picker, it must respond to the `kGetPrompt` request code. The Color Picker Manager sends this code to obtain the prompt string currently used by your color picker. A color picker responds to the `kGetPrompt` request

code by calling a color picker–defined subroutine (for example, `MyGetPrompt`) to handle the request.

```
pascal ComponentResult MyGetPrompt (
                    PickerStorageHndl storage,
                    Str255 prompt);
```

`storage` A handle to your color picker's global data.

`prompt` Your color picker's prompt string.

DESCRIPTION

Your `MyGetPrompt` function should return the current prompt for your color picker in the `prompt` parameter.

Your function should return `noErr` if successful, or an appropriate result code otherwise.

SEE ALSO

Listing 2-27 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to implement this function.

MySetPrompt

If you create a color picker, it must respond to the `kSetPrompt` request code. The Color Picker Manager sends this code to set the prompt string used by your color picker. A color picker responds to the `kSetPrompt` request code by calling a color picker–defined subroutine (for example, `MySetPrompt`) to handle the request.

```
pascal ComponentResult MySetPrompt (
                    PickerStorageHndl storage,
                    Str255 prompt);
```

`storage` A handle to your color picker's global data.

`prompt` The new prompt string for your color picker.

DESCRIPTION

Your `MySetPrompt` function should set the prompt for your color picker to the one specified in the `prompt` parameter.

Your function should return `noErr` if successful, or an appropriate result code otherwise.

SEE ALSO

Listing 2-28 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to implement this function.

MySetOrigin

If you create a color picker, it must respond to the `kSetOrigin` request code. The Color Picker Manager sends this code to inform your color picker that the window origin for the color picker has changed. A color picker responds to the `kSetOrigin` request code by calling a color picker–defined subroutine (for example, `MySetOrigin`) to handle the request.

```
pascal ComponentResult MySetOrigin (
                        PickerStorageHndl storage,
                        Point where);
```

`storage` A handle to your color picker's global data.

`where` The new window origin for the color picker.

DESCRIPTION

If your color picker maintains any information based on the local coordinate system of its dialog box, your `MySetOrigin` function should update that information.

The Color Picker Manager moves all dialog box items automatically in response to a new window origin for the dialog box, so it is not necessary for your color picker to move its items.

Your function should return `noErr` if successful, or an appropriate result code otherwise.

MyGetProfile

If you create a color picker, it must respond to the `kGetProfile` request code. For color-matching purposes, the Color Picker Manager sends this code to obtain the destination profile used by your color picker. A color picker responds to the `kGetProfile` request code by calling a color picker–defined subroutine (for example, `MyGetProfile`) to handle the request.

```
pascal CMProfileHandle MyGetProfile (PickerStorageHndl storage);
```

`storage` A handle to your color picker's global data.

DESCRIPTION

As its function result, your `MyGetProfile` function should return a handle to the destination ColorSync 1.0 profile used by your color picker.

SPECIAL CONSIDERATIONS

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" in *Advanced Color Imaging on the Mac OS* 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.

SEE ALSO

Listing 2-29 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to implement this function.

MySetProfile

If you create a color picker, it must respond to the `kSetProfile` request code. For color-matching purposes, the Color Picker Manager sends this code to set the destination profile used by your color picker. A color picker responds to the `kSetProfile` request code by calling a color picker–defined subroutine (for example, `MySetProfile`) to handle the request.

```
pascal ComponentResult MySetProfile (
                    PickerStorageHndl storage,
                    CMProfileHandle profile);
```

`storage` A handle to your color picker's global data.

`profile` A handle to a ColorSync 1.0 profile (that is, a `CMProfile` structure).

DESCRIPTION

Your `MySetProfile` function should set the destination profile to the one specified in the `profile` parameter.

Your function should return `noErr` if successful, or an appropriate result code otherwise.

SPECIAL CONSIDERATIONS

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" in *Advanced Color Imaging on the Mac OS* 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.

SEE ALSO

Listing 2-30 in the chapter “Color Picker Manager” in *Advanced Color Imaging on the Mac OS* illustrates how to implement this function.

MyGetEditMenuState

If you create a color picker, it must respond to the `kGetEditMenuState` request code. The Color Picker Manager sends this code to request information about the desired state of the Edit menu for your color picker. A color picker responds to the `kGetEditMenuState` request code by calling a color picker–defined subroutine (for example, `MyGetEditMenuState`) to handle the request.

```
pascal ComponentResult MyGetEditMenuState (
                        PickerStorageHndl storage,
                        MenuState *mState);
```

`storage` A handle to your color picker’s global data.

`mState` A `MenuState` structure, as described on page 2-20.

DESCRIPTION

In the `MenuState` structure pointed to in the `mState` parameter, your `MyGetEditMenuState` function should return information about your color picker’s Edit menu. The Color Picker Manager sets the Edit menu to this state.

Your function should return `noErr` if successful, or an appropriate result code otherwise.

SEE ALSO

Listing 2-31 in the chapter “Color Picker Manager” in *Advanced Color Imaging on the Mac OS* illustrates how to implement this function.

MyExtractHelpItem

If you create a color picker, it must respond to the `kExtractHelpItem` request code. The Color Picker Manager sends this code to obtain help messages or other help balloon characteristics from your color picker. A color picker responds to the `kExtractHelpItem` request code by calling a color picker–defined subroutine (for example, `MyExtractHelpItem`) to handle the request.

```
pascal ComponentResult MyExtractHelpItem (
                        PickerStorageHndl storage,
                        short itemNo,
                        short whichState,
                        HelpItemInfo *helpInfo);
```

`storage` A handle to your color picker's global data.

`itemNo` A number corresponding to the position of an item in the item list resource of the color picker's dialog box. The Help Manager is ready to display a help balloon for the item represented by this number.

`whichState` For menu items and items in alert or dialog boxes, the state of the item specified in the `itemNo` parameter. The following constants are used to represent the possible states: `kHMEnabledItem`, `kHMDisabledItem`, `kHMCheckedItem`, and `kHMOtherItem`.

`helpInfo` A `HelpItemInfo` structure, as described on page 2-33. In the `helpMessage` field of this structure, the Color Picker Manager passes the default help message in an `HMHelpMessage` structure; your `MyExtractHelpItem` function changes the help message by supplying a different `HMHelpMessage` structure (described in the chapter "Help Manager" in *Inside Macintosh: More Macintosh Toolbox*) for this field. The Color Picker Manager passes the default characteristics of the help balloon—that is, the value of the `options` element of its help resource, its tip location, its alternate rectangle, its tip function, and its variation code—in the rest of the fields of the `HelpItemInfo` structure, which your `MyExtractHelpItem` function can also change.

DESCRIPTION

Your `MyExtractHelpItem` function should return information about your color picker's help balloons. If your color picker has no help balloons, it should return the `noHelpForItem` result code, and the Help Manager will display the default message and characteristics of the help balloon.

Responding to Events in a Color Picker

This section describes the functions that your color picker should define for responding to events involving your color picker.

MyDrawPicker

If you create a color picker, it must respond to the `kDrawPicker` request code. The Color Picker Manager sends this code in response to an update event. A color picker responds to the `kDrawPicker` request code by calling a color picker–defined subroutine (for example, `MyDrawPicker`) to handle the request.

```
pascal ComponentResult MyDrawPicker (PickerStorageHndl storage);
```

`storage` A handle to your color picker's global data.

DESCRIPTION

Your `MyDrawPicker` function should redraw your color picker. The Color Picker Manager calls the Event Manager function `BeginUpdate` before sending the `kDrawPicker` request code, and the Color Picker Manager calls the Event Manager function `EndUpdate` after sending the `kDrawPicker` request code.

Your function should return `noErr` if successful, or an appropriate result code otherwise.

SEE ALSO

Listing 2-20 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to implement this function.

MyDoEvent

If you create a color picker, it must respond to the `kEvent` request code. The Color Picker Manager sends this 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 calling a color picker–defined subroutine (for example, `MyDoEvent`) to handle the request.

```
pascal ComponentResult MyDoEvent (
                    PickerStorageHndl storage,
                    EventData *data);
```

`storage` A handle to your color picker's global data.

`data` An `EventData` structure, as described on page 2-27.

DESCRIPTION

If your color picker needs to perform any event processing in addition to or instead of that normally performed by the Dialog Manager, your `MyDoEvent` function should perform it. The event is passed to your function in the event record pointed to in the `event` field of the `EventData` structure which, in turn, is pointed to in the `data` parameter.

In the `EventData` structure pointed to in the `data` parameter, your `MyDoEvent` function returns information about any event handling it performs. If your function handles the event, it should set the value of the `handled` field to `true`, in which case the Dialog Manager performs no additional handling of the event. Your function should set the `action` field to the particular action it performed. The `colorProc` field may point to an application-defined function that your color picker should call.

Your function should return `noErr` if successful, or an appropriate result code otherwise.

SEE ALSO

Listing 2-21 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to implement this function.

MyItemHit

If you create a color picker, it must respond to the `kItemHit` request code. The Color Picker Manager sends this code to inform your color picker of an event in one of its items. A color picker responds to the `kItemHit` request code by calling a color picker–defined subroutine (for example, `MyItemHit`) to handle the request.

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

`storage` A handle to your color picker's global data.

`data` An `ItemHitData` structure, as shown on page 2-31.

DESCRIPTION

Your `MyItemHit` function should respond to the event for the item reported in the `itemHit` field of the `ItemHitData` structure pointed to in the `data` parameter. (This item is passed by the Dialog Manager function `DialogSelect`.)

The `iMod` field of the `ItemHitData` structure informs your function of the action in the item. These actions are represented by the following constants:

```
enum ItemHitModifiers {
    kMouseDown,             /* mouse-down event on the 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;
```

In the `ItemHitData` structure pointed to in the `data` parameter, your `MyItemHit` function returns information about any event handling it performs. Your function should set the `action` field to the particular action it performed. The

`colorProc` field may contain a pointer to an application-defined function to handle color changes. This function, described in detail on page 2-60, should support the updating of colors in a document as the user selects them. Your color picker should call this function.

Your function should return `noErr` if successful, or an appropriate result code otherwise.

SEE ALSO

Listing 2-22 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to implement this function.

MyDoEdit

If you create a color picker, it must respond to the `kEdit` request code. The Color Picker Manager sends this code to inform your color picker that the user has chosen one of the edit commands from the Edit menu (or the user has typed a Command-key equivalent). A color picker responds to the `kEdit` request code by calling a color picker–defined subroutine (for example, `MyDoEdit`) to handle the request.

```
pascal ComponentResult MyDoEdit (
                      PickerStorageHndl storage,
                      EditData *data);
```

`storage` A handle to your color picker's global data.

`data` An `EditData` structure, as described on page 2-29.

DESCRIPTION

If your color picker needs to handle an editing command instead of allowing the Dialog Manager to handle it, your `MyDoEdit` function should perform it. For example, because the Dialog Manager does not handle the Undo command, your `MyDoEdit` function can handle it instead. The editing command is passed to your function in the field `theEdit` of the `EditData` structure pointed to in the `data` parameter.

If your function handles the command, it should set the `handled` field of the `EditData` structure to `true`, in which case the Dialog Manager performs no additional processing of the associated event. If your function sets the `handled` field to `false`, then the Color Picker Manager sends your color picker the `kItemHit` request code with the appropriate information regarding the event in the editable-text item.

Your function should return `noErr` if successful, or an appropriate result code otherwise.

SEE ALSO

Listing 2-23 in the chapter "Color Picker Manager" in *Advanced Color Imaging on the Mac OS* illustrates how to implement this function.

Result Codes

`firstPickerError`	**–4000**
`invalidPickerType`	**–4000**
`requiredFlagsDontMatch`	**–4001**
`pickerResourceError`	**–4002**
`cantLoadPicker`	**–4003**
`cantCreatePickerWindow`	**–4004**
`cantLoadPackage`	**–4005**
`pickerCantLive`	**–4006**
`colorSyncNotInstalled`	**–4007**
`badProfileError`	**–4008**
`noHelpForItem`	**–4009**

ColorSync Manager Reference for Applications and Device Drivers

The ColorSync Manager Constants and Data Structures 3-5
Constants for Profile Location Type 3-5
Constants for ColorSync Manager Gestalt Selectors and Responses 3-7
Profile Classes 3-8
Signature of the Apple-Supplied Color Management Module 3-9
Commands for Calling the Caller-Supplied ColorSync Data Transfer Functions 3-9
Picture Comment IDs for Profiles and Color Matching 3-10
Picture Comment Selectors for the cmComment ID 3-11
Color Space Signatures 3-13
Color Packing for Color Spaces 3-14
Color Spaces 3-15
Rendering Intent Values for Version 2.0 Profiles 3-19
Function Selectors for Color-Conversion-Component Functions 3-20
Operation Codes Used With PrGeneral Function 3-22
Color Conversion Component Version 3-22
The ColorSync Manager Element Tags and Their Signatures for Version 1.0 Profiles 3-22
Profile Location Union 3-23
Profile Location Structure 3-24
File Specification for a File-Based Profile 3-24
Handle Specification for a Memory-Based Profile 3-25
Pointer Specification for a Memory-Based Profile 3-25
Apple Profile Header 3-26
Profile 2.0 Header Structure for the ColorSync Manager 3-26
Concatenated Profile Set Structure 3-30
Color World Information Record 3-31

This document was created with FrameMaker 4.0.4

Color Management Module (CMM) Information Record Structure 3-32
Profile Search Record 3-33
XYZ Color Component Values 3-35
XYZ Color Value 3-35
Fixed XYZ Color Value 3-35
L*a*b* Color Value 3-36
L*u*v* Color Value 3-36
Yxy Color Value 3-37
RGB Color Value 3-37
HLS Color Value 3-37
HSV Color Value 3-38
CMYK Color Value 3-38
CMY Color Value 3-39
HiFi Color Values 3-39
Gray Color Value 3-39
The Color Union 3-40
The ColorSync Manager Bitmap 3-42
Profile Reference 3-43
Profile Search Result Reference 3-44
High-Level Color-Matching-Session Reference 3-44
Color World Reference 3-44
TEnableColorMatchingBlk 3-45
Profile Header for ColorSync 1.0 3-45
PostScript Color Rendering Dictionary (CRD) Virtual Memory Size Tag Structure 3-48
The ColorSync Manager Functions 3-49
Accessing Profile Files 3-50
CMOpenProfile 3-50
CMCloseProfile 3-51
CMUpdateProfile 3-52
CMNewProfile 3-53
CMCopyProfile 3-55
CMGetProfileLocation 3-56
CMValidateProfile 3-57
CMFlattenProfile 3-58
CMUnflattenProfile 3-59
Accessing Profile Elements 3-60
CMProfileElementExists 3-61

CMCountProfileElements 3-61
CMGetProfileElement 3-62
CMGetProfileHeader 3-64
CMGetPartialProfileElement 3-65
CMGetIndProfileElementInfo 3-66
CMGetIndProfileElement 3-67
CMSetProfileElementSize 3-69
CMSetPartialProfileElement 3-70
CMSetProfileElement 3-71
CMSetProfileHeader 3-72
CMSetProfileElementReference 3-73
CMRemoveProfileElement 3-73
CMGetScriptProfileDescription 3-74
Matching Colors Using the High-Level Functions 3-75
NCMBeginMatching 3-75
CMEndMatching 3-77
CMEnableMatchingComment 3-77
Using Embedded Profiles With QuickDraw 3-78
NCMDrawMatchedPicture 3-78
NCMUseProfileComment 3-79
Matching Colors Using the Low-Level Functions Without QuickDraw 3-80
NCWNewColorWorld 3-81
CWConcatColorWorld 3-82
CWNewLinkProfile 3-84
CWDisposeColorWorld 3-86
CMGetCWInfo 3-87
CWMatchPixMap 3-88
CWCheckPixMap 3-90
CWMatchBitmap 3-92
CWCheckBitMap 3-95
CWMatchColors 3-97
CWCheckColors 3-98
Assigning and Accessing the System Profile File 3-99
CMSetSystemProfile 3-99
CMGetSystemProfile 3-100
Searching External Profiles 3-101
CMNewProfileSearch 3-101

CMUpdateProfileSearch 3-102
CMDisposeProfileSearch 3-104
CMSearchGetIndProfile 3-104
CMSearchGetIndProfileFileSpec 3-105
Converting Between Color Spaces 3-106
CMXYZToLab 3-108
CMLabToXYZ 3-109
CMXYZToLuv 3-111
CMLuvToXYZ 3-112
CMXYZToYxy 3-113
CMYxyToXYZ 3-114
CMXYZToFixedXYZ 3-116
CMFixedXYZToXYZ 3-117
CMRGBToHLS 3-118
CMHLSToRGB 3-120
CMRGBToHSV 3-121
CMHSVToRGB 3-122
CMRGBToGray 3-124
PostScript Color-Matching Support Functions 3-125
CMGetPS2ColorSpace 3-125
CMGetPS2ColorRenderingIntent 3-126
CMGetPS2ColorRendering 3-128
CMGetPS2ColorRenderingVMSize 3-129
Locating the ColorSync Profiles Folder 3-130
CMGetColorSyncFolderSpec 3-130
Application-Defined Functions for the ColorSync Manager 3-131
MyColorSyncDataTransfer 3-131
MyCMBitmapCallBackProc 3-134
MyCMProfileFilterProc 3-136
Result Codes 3-138

This reference document describes the constants, data structures, and functions defined for your application's use by the ColorSync Manager. To support the ColorSync Manager in your applications and device drivers, use the ColorSync Manager API.

However, the ColorSync Manager provides backward compatibility with ColorSync 1.0 for those applications and device drivers written to the ColorSync 1.0 API. Your application, written to the ColorSync Manager API, can match, convert, and color-check colors using version 2.0 profiles or, when necessary, a combination of version 2.0 profiles and ColorSync 1.0 profiles. For a description of the backward compatibility support provided by the ColorSync Manager, see the appendix "ColorSync Manager Backward Compatibility"in *Advanced Color Imaging on the Mac OS.*

The ColorSync Manager Constants and Data Structures

This section describes the constants and data structures defined for your application's use with version 2.0 of the ColorSync Manager API.

Constants for Profile Location Type

A ColorSync profile is typically stored in a disk file whose location your application provides using a data structure of type `CMProfileLocation`, described in "Profile Location Structure" on page 3-24. The profile location data structure contains a `u` field whose union holds a file specification for a profile that is disk-file based. However, to accommodate special requirements, a ColorSync profile that you copy can be stored in nonrelocatable memory and a ColorSync profile that you open or create can be stored relocatable memory. For these special requirements, your application can provide a handle or pointer specification in the `u` field of the structure `CMProfileLocation`. Additionally, your application can create a new or duplicate temporary profile. For example, you can use a temporary profile for a color-matching session and the profile will not be saved after the session. For this case, the ColorSync Manager allows you to specify the profile location as having no specific location; that is, it is not based in memory or on disk.

You use a data structure of type `CMProfileLocation` to identify a profile's location when

This document was created with FrameMaker 4.0.4

- Your application calls the `CMOpenProfile` function to obtain a reference to the profile
- To specify the location for a newly created or duplicate profile when your application calls the `CMNewProfile`, `CWNewLinkProfile`, or `CMCopyProfile` functions.

Your application identifies the type of data the `CMProfileLocation` `u` field holds—a file specification, a handle, or a pointer—in the `CMProfileLocation` structure's `locType` field. You use the constants defined by the following enumeration to identify the location type.

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

Constant descriptions

`cmNoProfileBase`
: The profile is temporary. It will not persist in memory after its use for a color session. You can specify this type of profile location with the `CMNewProfile` and the `CMCopyProfile` functions.

`cmFileBasedProfile`
: The profile is disk-file based and the `CMProfLoc` union, described beginning on page 3-23, holds a structure of type `CMFileLocation` identifying the profile file. For a description of the `CMFileLocation` type definition, see "File Specification for a File-Based Profile" on page 3-24. You can specify this type of profile location with the `CMOpenProfile`, `CMNewProfile`, `CMCopyProfile`, and `CMNewLinkProfile` functions.

`cmHandleBasedProfile`
: The profile is relocatable-memory based and the `CMProfLoc` union, described beginning on page 3-23, holds a handle to the profile in a structure of type `CMHandleLocation`. For a description of the `CMHandleLocation` type definition, see "Handle Specification for a Memory-Based Profile" on page 3-25.You can specify this type of profile location with

the `CMOpenProfile`, `CMNewProfile`, and `CMCopyProfile` functions.

`cmPtrBasedProfile`
: The profile is nonrelocatable-memory based and the `CMProfLoc` union, described beginning on page 3-23, holds a pointer to the profile in a structure of type `CMPtrLocation`. For a description of the `CMPtrLocation` type definition, see “Pointer Specification for a Memory-Based Profile” on page 3-25. You can specify this type of profile location with the `CMOpenProfile` function only.

Constants for ColorSync Manager Gestalt Selectors and Responses

These enumerations define the constants for the selectors that you use with the `Gestalt` function to test if the ColorSync Manager is available. To test for the availability of the ColorSync Manager on a 68K-based Mac system, you use the `Gestalt` function with the `gestaltColorMatchingVersion` selector. To test for the availability of the ColorSync Manager on a PowerPC-based system, you use the `Gestalt` function with the `gestaltColorMatchingAttr` selector. The enumerations also define the constants for response values.

The constants `gestaltColorSync10` and `gestaltColorSync11` are returned for ColorSync versions 1.0 to 1.0.3. The constant `gestaltColorSync11` indicates low-level matching only.

```
enum {
    gestaltColorMatchingAttr         = 'cmta',
    gestaltHighLevelMatching         = 0,
    gestaltColorMatchingLibLoaded    = 1
};

enum {
    gestaltColorMatchingVersion      = 'cmtc',
    gestaltColorSync10               = 0x0100,
    gestaltColorSync11               = 0x0110,
    gestaltColorSync104              = 0x0104,
    gestaltColorSync105              = 0x0105,
    gestaltColorSync20               = 0x0200
};
```

Profile Classes

The ColorSync Manager supports six classes, or types, of profiles. Three of the profile classes define device profiles for different types of devices: input, output, and display devices. A device profile describes the characteristics of a particular device and shows how color is to be converted and matched for the device in a particular state.

The other three profile classes include definitions for an abstract profile, a color space profile, and a device-linked profile. A color space profile contains the data necessary to translate color values expressed in one color space to another, for example, CIE into RGB, or vice versa, as necessary for color matching. An abstract profile allows applications to perform special color effects independent of the devices on which the effects are rendered. A device-linked profile combines multiple profiles, such as various device profiles and color space profiles associated with the creation and editing of an image. A profile creator specifies the profile type in the profile header's `profileClass` field. (For a description of the version 2.0 profile header, see "Profile 2.0 Header Structure for the ColorSync Manager" on page 3-26.) The following enumeration defines the profile type signatures:

```
enum {
    cmInputClass            = 'scnr',
    cmDisplayClass          = 'mntr',
    cmOutputClass           = 'prtr',
    cmLinkClass             = 'link',
    cmAbstractClass         = 'abst',
    cmColorSpaceClass       = 'spac'
    };
```

Constant descriptions

`cmInputClass` An input device profile defined for a scanner with a signature of `'scnr'`.

`cmDisplayClass` A display device profile defined for a monitor with a signature of `'mntr'`.

`cmOutputClass` An output device profile defined for a printer with a signature of `'prtr'`.

`cmLinkClass` A device-linked profile with a signature of `'link'`.

`cmAbstractClass` An abstract profile with a signature of `'abst'`.

`cmColorSpaceClass`
A color space profile with a signature of `'spac'`.

Signature of the Apple-Supplied Color Management Module

Apple provides a robust color management module (CMM) with the ColorSync Manager that is installed as part of the ColorSync extension. The ColorSync Manager uses this CMM as the default when a profile specifies a preferred CMM that is unavailable or unable to perform a requested function. The Apple-supplied default CMM supports all the required and optional functions that make up the ColorSync Manager API for CMMs. For a description of the CMM functions, see "ColorSync Manager Reference for Color Management Modules." Device manufacturers and peripheral developers can provide their own CMMs or use the one Apple supplies.

To specify explicitly that the Apple-supplied CMM is to be used, set the `CMMType` field of the profile header to the `'appl'` signature defined by the following enumeration. For a description of the `CM2Header`, see "Profile 2.0 Header Structure for the ColorSync Manager" on page 3-26. For a description of the `CMHeader`, see "Profile Header for ColorSync 1.0" on page 3-45.

```
enum {
    kDefaultCMMSignature= 'appl'
    };
```

Commands for Calling the Caller-Supplied ColorSync Data Transfer Functions

When your application calls the `CMFlattenProfile` function, the `CMUnflattenProfile` function, or the PostScript-related functions, the selected CMM—and also the ColorSync dispatcher for the `CMUnflattenProfile` function—calls the flatten function you supply to transform profile data. Your function is called with one of the commands defined by this enumeration.

Your application provides a pointer to your ColorSync data transfer function as a parameter to the functions. The dispatcher or the CMM calls your calling program-supplied ColorSync data transfer function passing the command in the `command` parameter. For more information on the flatten function, see "MyColorSyncDataTransfer" on page 3-131.

```
enum {
        openReadSpool = 1,
        openWriteSpool,
        readSpool,
        writeSpool,
        closeSpool
};
```

Constant descriptions

`openReadSpool` Directs the function to begin the process of reading data.

`openWriteSpool` Directs the function to begin the process of writing data.

`readSpool` Directs the function to read the number of bytes specified by the `MyColorSyncDataTransfer` function's `size` parameter.

`writeSpool` Directs the function to write the number of bytes specified by the `MyColorSyncDataTransfer` function's `size` parameter.

`closeSpool` Directs the function to complete the data transfer.

Picture Comment IDs for Profiles and Color Matching

The ColorSync Manager defines the five picture comments for turning on and off use of embedded profiles and performing color matching within drawing code sent to an output device. Your application uses the QuickDraw `PicComment` function, described in *Inside Macintosh: Imaging With QuickDraw*, to specify these picture comments when turning use of embedded profiles on and off or turning color matching on and off. The `PicComment` function's `kind` parameter specifies the kind of picture comment.

IMPORTANT

When you want to terminate use of the currently effective embedded profile, you should do so explicitly by specifying a picture comment of kind `cmEndProfile`. This terminates use of the current profile and instates use of the system profile. If you do not include this picture comment, the currently effective profile remains in effect. This can cause problems if another picture follows that isn't preceded by a profile because the intention is to use the system profile for that picture. In this case, the currently effective profile will be used, not the system profile. ▲

```
enum {
    cmBeginProfile      = 220,
    cmEndProfile        = 221,
    cmEnableMatching    = 222,
    cmDisableMatching   = 223,
    cmComment           = 224
    };
```

Constant descriptions

`cmBeginProfile`
: Indicates the beginning of a version 1.0 profile to be embedded.

`cmEndProfile`
: Signals end of the use of an embedded version 2.0 or 1.0 profile.

`cmEnableMatching`
: Turns on color matching for either the ColorSync Manager 2.0 or 1.0. Do not nest `cmEnableMatching` and `cmDisableMatching` pairs.

`cmDisableMatching`
: Turns off color matching for either the ColorSync Manager 2.0 or 1.0. Do not nest `cmEnableMatching` and `cmDisableMatching` pairs. After the ColorSync Manager encounters this comment, it ignores all ColorSync-related picture comments until it encounters the next `cmEnableMatching` picture comment. At this point, the last most recently used profile is reinstated.

`cmComment`
: Provides information about a 2.0 embedded profile. This picture comment is followed by a 4-byte selector further identifying whether the beginning of a version 2.0 embedded profile follows, more of the profile data follows, or no profile data follows because the end has been reached. See "Picture Comment Selectors for the cmComment ID" on page 3-11 for more information on the selectors.

Picture Comment Selectors for the cmComment ID

To embed a version 2.0 profile in a picture destined for display on another system or an output device such as a printer, your application uses the QuickDraw `PicComment` function specifying a picture comment `kind` value of

`cmComment` (224) followed by a 4-byte selector describing the data in the picture comment.

A profile may exceed the Quickdraw `PicComments` 32 KB size limit. To accommodate large profiles, your application can use an ordered series of picture comments to embed the profile.

Your application specifies one of the 4-byte selectors defined by the following enumeration after the `cmComment kind` value to identify the beginning and continuation of profile data and to signal the end of it. For a description of how to use the `PicComment` function to embed a profile, see the chapter "Developing ColorSync-Supportive Applications" in *Advanced Color Imaging on the Mac OS.*

```
enum {
    cmBeginProfileSel           = 0,
    cmContinueProfileSel        = 1,
    cmEndProfileSel             = 2
    };
```

Constant descriptions

`cmBeginProfileSel`
: Identifies the beginning of a version 2.0 profile data. The amount of profile data you can specify is limited to 32K minus 4 bytes for selector 0.

`cmContinueProfileSel`
: Identifies the continuation of version 2.0 profile data. The amount of profile data you can specify is limited to 32K minus 4 bytes for selector 1. You can use this selector repeatedly until all the profile data is embedded.

`cmEndProfileSel`
: Signals the end of version 2.0 profile data, no more data follows. Even if the amount of profile data embedded does not exceed 32K minus 4 bytes for the selector and your application did not use selector 1, you must terminate the process with selector 2. Note that this selector has a behavior that is different from the `cmEndProfile` picture comment described in "Picture Comment IDs for Profiles and Color Matching," beginning on page 3-10.

Color Space Signatures

A ColorSync version 2.0 profile header contains a `dataColorSpace` field that carries the signature of the data color space in which the color values of colors in an image using the profile are expressed. This enumeration defines the signatures for the color spaces supported by ColorSync for version 2.0 profiles.

```
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'
};
```

Constant descriptions

`cmXYZData` The XYZ data color space with a signature of `'XYZ '`.

`cmLabData` The L*a*b* data color space with a signature of `'Lab '`.

`cmLuvData` The L*u*v* data color space with a signature of `'Luv '`.

`cmYxyData` The Yxy data color space with a signature of `'Yxy '`.

`cmRGBData` The RGB data color space with a signature of `'RGB '`.

`cmGrayData` The Gray data color space with a signature of `'GRAY'`.

`cmHSVData` The HSV data color space with a signature of `'HSV '`.

`cmHLSData` The HLS data color space with a signature of `'HLS '`.

`cmCMYKData` The CMYK data color space with a signature of `'CMYK'`.

`cmCMYData` The CMY data color space with a signature of `'CMY '`.

`cmMCH5Data` The five-channel multichannel (HiFi) data color space with a signature of `'MCH5'`.

`cmMCH6Data` The six-channel multichannel (HiFi) data color space with a signature `'MCH6'`.

`cmMCH7Data` The seven-channel multichannel (HiFi) data color space with a signature `'MCH7'`.

`cmMCH8Data` The eight-channel multichannel (HiFi) data color space with a signature `'MCH8'`.

Color Packing for Color Spaces

The ColorSync Manager bitmap `CMBitMap` data type includes a field that identifies the color space in which the color values of the bitmap image are expressed. The following enumeration defines the types of packing used for a color space's storage format. The enumeration also defines an alpha channel that can be added as a component of a color value to define the degree of opacity or transparency of a color. These constants are combined with data color space constants in the enumeration described in "Color Spaces" on page 3-15 to create values that identify a bitmap's color space. Your application does not specify these constants directly.

```
enum {
    cmNoColorPacking        = 0x0000,
    cmAlphaSpace            = 0x0080,
    cmWord5ColorPacking     = 0x0500,
    cmLong8ColorPacking     = 0x0800,
    cmLong10ColorPacking    = 0x0a00,
    cmAlphaFirstPacking     = 0x1000,
    cmOneBitDirectPacking   = 0x0b00
    };
```

Constant descriptions

`cmNoColorPacking`
This constant is not used for ColorSync bitmaps.

`cmAlphaSpace` An alpha channel component is added to the color value.

`cmWord5ColorPacking`
Each color component value is stored as 5 bits.

`cmLong8ColorPacking`
Each color component value is stored as 8 bits.

`cmLong10ColorPacking`
: Each color component value is stored as 10 bits.

`cmAlphaFirstPacking`
: An alpha channel is added to the color value as its first component.

`cmOneBitDirectPacking`
: One bit is used as the pixel format. This storage format is used by the resulting bitmap pointed to by the `resultBitMap` field of the `CWCheckBitMap` function, described on page 3-95, in which the bitmap must be only 1 bit deep.

Color Spaces

The `CMBitmap` data type defines a bitmap for an image whose colors may be matched using the `CWMatchBitmap` function described on page 3-92 or color-checked using the `CWCheckBitMap` function described on page 3-95.

The `space` field of a `CMBitmap` type definition identifies the color space in which the colors of the bitmap image are specified. A color space is characterized by a number of components or dimensions with each component carrying a numeric value that together comprise the color value. A color space also specifies the format in which the color value is stored. For bitmaps in which color values are packed, the `space` field of the `CMBitmap` data type holds a constant that defines the color space and the packing format.

The following enumeration defines the constants representing the various color spaces and packing formats in which color values of an image represented by a bitmap of type `CMBitmap` may be specified. All of these constants include a packing format except `cmGraySpace`. The packing format for `cmRGBASpace` is 64 bits long.

For color matching to complete successfully using the `CWMatchBitmap` function, the color space specified in the `CMBitmap` data type's `space` field must match the color space specified in the profile's `dataColorSpace` field. These source bitmap and source profile values must match each other and the destination bitmap and destination profile values must match each other. For color checking to complete successfully using the `CWCheckBitMap` function, the source profile's `dataColorSpace` field value and the `space` field value of the source bitmap must specify the same color space. These functions will complete successfully as long as the color spaces are the same without regard for the packing format specified by the bitmap.

```
enum {
    cmNoSpace           = 0,
    cmRGBSpace,
    cmCMYKSpace,
    cmHSVSpace,
    cmHLSSpace,
    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 + cmHLSSpace,
    cmYXY32Space            = cmLong10ColorPacking + cmYXYSpace,
    cmXYZ32Space            = cmLong10ColorPacking + cmXYZSpace,
    cmLUV32Space            = cmLong10ColorPacking + cmLUVSpace,
    cmLAB32Space            = cmLong10ColorPacking + cmLABSpace,
    cmGamutResult1Space     = cmOneBitDirectPacking +
                                cmGamutResultSpace
};
```

Constant descriptions

`cmNoSpace` The ColorSync Manager does not use this constant.

`cmRGBSpace` An RGB color space composed of red, green, and blue components. A bitmap never uses this constant alone. Instead, this color space is always combined with a packing format describing the amount of storage per component.

`cmCMYKSpace` A CMYK color space composed of cyan, magenta, yellow, and black. A bitmap never uses this constant alone.

Instead, this color space is always combined with a packing format describing the amount of storage per component.

`cmHSVSpace` An HSV color space composed of hue, saturation, and value components. A bitmap never uses this constant alone. Instead, this color space is always combined with a packing format describing the amount of storage per component.

`cmHLSSpace` An HLS color space composed of hue, lightness, and saturation components. A bitmap never uses this constant alone. Instead, this color space is always combined with a packing format describing the amount of storage per component.

`cmYXYSpace` A Yxy color space composed of Y, x, and y components. A bitmap never uses this constant alone. Instead, this color space is always combined with a packing format describing the amount of storage per component.

`cmXYZSpace` An XYZ color space composed of X, Y, and Z components. A bitmap never uses this constant alone. Instead, this color space is always combined with a packing format describing the amount of storage per component.

`cmLUVSpace` An L*u*v* color space composed of L*, u*, and v* components. A bitmap never uses this constant alone. Instead, this color space is always combined with a packing format describing the amount of storage per component.

`cmLABSpace` An L*a*b* color space composed of L*, a*, b* components. A bitmap never uses this constant alone. Instead, this color space is always combined with a packing format describing the amount of storage per component.

`cmReservedSpace1`
This field is reserved for use by QuickDraw GX.

`cmGraySpace` A luminance color space with a single component, gray.

`cmReservedSpace2` This field is reserved for use by QuickDraw GX.

`cmGamutResultSpace`
A color space used for the resulting bitmap pointed to by the `resultBitMap` field of the `CWCheckBitMap` function, described on page 3-95. A bitmap never uses this constant

alone. Instead, the constant `cmGamutResult1Space` is used, combining this value and `cmOneBitDirectPacking` to define a bitmap that is 1 bit deep.

`cmRGBASpace` — An RGB color space composed of red, green, and blue color value components and an alpha channel component. A bitmap never uses this constant alone. Instead, this constant is used to indicate the presence of an alpha channel in combination with `cmLong8ColorPacking` to indicate 8-bit packing format and `cmAlphaFirstPacking` to indicate the position of the alpha channel as the first component. The storage size for a color value expressed in this color space is 64 bits.

`cmGrayASpace` — A luminance color space with two components, a gray component followed by an alpha channel component. Each component value is 16 bits.

`cmRGB16Space` — An RGB color space composed of red, green, and blue components whose values are packed with 5 bits of storage per component. The storage size for a color value expressed in this color space is 16 bits, with the high-order bit not used.

`cmRGB32Space` — An RGB color space composed of red, green, and blue components whose values are packed with 8 bits of storage per component. The storage size for a color value expressed in this color space is 32 bits, with bits 24–31 not used.

`cmARGB32Space` — An RGB color space composed of red, green, and blue color value components preceded by an alpha channel component whose values are packed with 8 bits of storage per component. The storage size for a color value expressed in this color space is 32 bits.

`cmCMYK32Space` — A CMYK color space composed of cyan, magenta, yellow, and black components whose values are packed with 8 bits of storage per component. The storage size for a color value expressed in this color space is 32 bits.

`cmHSV32Space` — An HSV color space composed of hue, saturation, and value components whose values are packed with 10 bits of storage per component. The storage size for a color value expressed in this color space is 32 bits, with the high-order 2 bits not used.

`cmHLS32Space` An HLS color space composed of hue, lightness, and saturation components whose values are packed with 10 bits of storage per component. The storage size for a color value expressed in this color space is 32 bits, with the high-order 2 bits not used.

`cmYXY32Space` A Yxy color space composed of Y, x, and y components whose values are packed with 10 bits of storage per component. The storage size for a color value expressed in this color space is 32 bits, with the high-order 2 bits not used.

`cmXYZ32Space` An XYZ color space composed of X, Y, and Z components whose values are packed with 10 bits per component. The storage size for a color value expressed in this color space is 32 bits, with the high-order 2 bits not used.

`cmLUV32Space` An L*u*v* color space composed of L*, u*, and v* components whose values are packed with 10 bits per component. The storage size for a color value expressed in this color space is 32 bits, with the high-order 2 bits not used.

`cmLAB32Space` An L*a*b* color space composed of L*, a*, and b* components whose values are packed with 10 bits per component. The storage size for a color value expressed in this color space is 32 bits, with the high-order 2 bits not used.

`cmGamutResult1Space`

A gamut result color space used for the resulting bitmap pointed to by the `resultBitMap` field of the `CWCheckBitMap` function, described on page 3-95, with 1-bit direct packing.

Rendering Intent Values for Version 2.0 Profiles

The rendering intent specified by a profile controls the approach a CMM uses to translate the colors of an image to the color gamut of a destination device. Version 2.0 profiles support four types of rendering intents. Your application can set the intent, for example, based on a user's choice of the best approach for rendering an image. The following enumeration defines the possible rendering intents:

```
enum {
    cmPerceptual            = 0,
    cmRelativeColorimetric  = 1,
    cmSaturation            = 2,
    cmAbsoluteColorimetric  = 3
};
```

Constant descriptions

`cmPerceptual` All the colors of a given gamut may be scaled to fit within another gamut. This intent is best suited to realistic images, such as photographic images.

`cmRelativeColorimetric`
The colors that fall within the gamuts of both devices are left unchanged. This intent is best suited to logo images.

`cmSaturation`
The relative saturation of colors is maintained from gamut to gamut. This intent is best suited to bar graphs and pie charts in which the actual color displayed is less important than its vividness.

`cmAbsoluteColorimetric`
This approach is based on a device-independent color space in which the result is an idealized print viewed on a ideal type of paper having a large dynamic range and color gamut.

Function Selectors for Color-Conversion-Component Functions

The ColorSync Manager defines the following color-conversion-component function selectors used for the color conversion functions supported by the color conversion component. Your application does not use these selectors. Your application uses the color conversion functions, described beginning on page 3-106, to call the color conversion component to convert color values between color spaces within the same base families.

```
enum {
    kCMXYZToLab        = 0,
    kCMLabToXYZ        = 1,
    kCMXYZToLuv        = 2,
    kCMLuvToXYZ        = 3,
```

```
    kCMXYZToYxy         = 4,
    kCMYxyToXYZ         = 5,
    kCMRGBToHLS         = 6,
    kCMHLSToRGB         = 7,
    kCMRGBToHSV         = 8,
    kCMHSVToRGB         = 9,
    kCMRGBToGRAY        = 10,
    kCMXYZToFixedXYZ    = 11,
    kCMFixedXYZToXYZ    = 12
};
```

Constant descriptions

`kCMXYZToLab` Selector for the `CMXYZToLab` function described on page 3-108.

`kCMLabToXYZ` Selector for the `CMLabToXYZ` function described on page 3-109.

`kCMXYZToLuv` Selector for the `CMXYZToLuv` function described on page 3-111.

`kCMLuvToXYZ` Selector for the `CMLuvToXYZ` function described on page 3-112.

`kCMXYZToYxy` Selector for the `CMXYZToYxy` function described on page 3-113.

`kCMYxyToXYZ` Selector for the `CMYxyToXYZ` function described on page 3-114.

`kCMRGBToHLS` Selector for the `CMRGBToHLS` function described on page 3-118.

`kCMHLSToRGB` Selector for the `CMHLSToRGB` function described on page 3-120.

`kCMRGBToHSV` Selector for the `CMRGBToHSV` function described on page 3-121.

`kCMHSVToRGB` Selector for the `CMHSVToRGB` function described on page 3-122.

`kCMRGBToGRAY` Selector for the `CMRGBToGRAY` function described on page 3-124.

`kCMXYZToFixedXYZ` Selector for the `CMXYZToFixedXYZ` function described on page 3-116.

`kCMFixedXYZToXYZ` Selector for the `CMFixedXYZToXYZ` function described on page 3-117.

Operation Codes Used With PrGeneral Function

This enumeration defines operation codes used with the `PrGeneral` function to enable or disable color matching and, for ColorSync 1.0, to register a profile with the profile responder or remove the profile's registration. For information on the `PrGeneral` function, see *Inside Macintosh: Imaging With QuickDraw.*

```
enum {
    enableColorMatchingOp   = 12,
    registerProfileOp       = 13
};
```

Color Conversion Component Version

This enumeration defines the color conversion component version:

```
enum {
    CMConversionInterfaceVersion = 1
    };
```

The ColorSync Manager Element Tags and Their Signatures for Version 1.0 Profiles

The ICC version 2.0 profile format differs from the version 1.0 profile format. Your application cannot use the ColorSync Manager API to update a version 1.0 profile or search for version 1.0 profiles. However, your application can use the remaining ColorSync Manager API that pertains to profiles with version 1.0 profiles.

Your application can open a version 1.0 profile using the `CMOpenProfileFile` function, obtain the version 1.0 profile header using the `CMGetProfileHeader` function, and access version 1.0 profile elements using the `CMGetProfileElement` function.

To make this possible, the ColorSync Manager API includes support for the version 1.0 profile header structure and synthesizes tags to allow you to access four 1.0 elements outside the version 1.0 profile header. The following enumeration defines these tags:

```
enum {
        cmCS1ChromTag       = 'chrm',
        cmCS1TRCTag         = 'trc ',
```

```
        cmCS1NameTag        = 'name',
        cmCS1CustTag        = 'cust'
};
```

Constant descriptions

`'chrm'` The tag signature for the profile chromaticities tag whose element data specifies the XYZ chromaticities for the six primary and secondary colors (red, green, blue, cyan, magenta, and yellow).

`'trc '` Profile response data for the associated device.

`'name'` The tag signature for the profile name string. This is an international string consisting of a Macintosh script code followed by a 63-byte text string identifying the profile.

`'cust'` Private data for a custom CMM.

Profile Location Union

In most cases, a ColorSync version 2.0 profile is stored in a disk file. However, to support special requirements, a profile can also be located in memory. You use a union of type `CMProfLoc` to identify the location of a profile. You specify the union in the `u` field of the `CMProfileLocation` data type. Your application passes a `CMProfileLocation` structure to the function when it calls the `CMOpenProfile` function to identify the location of a profile or the `CMNewProfile`, `CWNewLinkProfile`, or `CMCopyProfile` functions to specify the location for a newly created profile.

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

Field descriptions

`fileLoc` A file system specification record of type `CMFileLocation` that tells the location of the profile disk file. For a description of the `CMFileLocation` data structure, see "File Specification for a File-Based Profile" on page 3-24.

`handleLoc` A data structure of type `CMHandleLocation` containing a handle that indicates the location of a profile in relocatable

memory. For a description of the `CMHandleLocation` data structure, see “Handle Specification for a Memory-Based Profile” on page 3-25.

`ptrLoc` A data structure of type `CMPtrLocation` holding a pointer that points to a profile in nonrelocatable memory. For a description of the `CMPtrLocationPtr` data structure, see “Pointer Specification for a Memory-Based Profile” on page 3-25.

Profile Location Structure

Your application passes a profile location structure of type `CMProfileLocation` to a function when it calls the `CMOpenProfile` function to identify the location of a profile or the `CMNewProfile`, `CWNewLinkProfile`, or `CMCopyProfile` functions to specify the location for a newly created or duplicate profile.

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

Field descriptions

`locType` The type of data structure the `u` field’s `CMProfLoc` union holds — a file specification, a handle, or a pointer. To specify the type, you use the constants defined in the enumeration described in “Constants for Profile Location Type” on page 3-5.

`u` A union of type `CMProfLoc` identifying the profile location. For a description of the `CMProfLoc` union, see “Profile Location Union” on page 3-23.

File Specification for a File-Based Profile

Your application uses the `CMFileLocation` structure to provide a file specification for a profile stored in a disk file. You provide the file specification structure in the `CMProfLoc` union of the `CMProfileLocation` structure’s `u` field to tell the location of an existing profile or where a profile is to be created.

```
struct CMFileLocation {
    FSSpec   spec;
};
```

Field description

`spec` A file system specification structure of type `FSSpec` that tells the location of the profile file. A file specification structure includes the volume reference number, the directory ID of the parent directory, and the filename or directory name. For a description of the `FSSpec` data structure, see *Inside Macintosh: Files.*

Handle Specification for a Memory-Based Profile

Your application uses the `CMHandleLocation` structure to provide a handle specification for a profile stored in relocatable memory. You provide the handle specification structure in the `CMProfLoc` union of the `CMProfileLocation` structure's `u` field to indicate an existing profile or where a profile is to be created.

```
struct CMHandleLocation {
    Handle   h;
};
```

Field description

`h` A data structure of type `Handle` containing a handle that indicates the location of a profile in memory. For a description of the `Handle` data structure, see *Inside Macintosh: Memory.*

Pointer Specification for a Memory-Based Profile

Your application uses the `CMPtrLocation` structure to provide a pointer specification for a profile stored in nonrelocatable memory. You provide the pointer specification structure in the `CMProfLoc` union of the `CMProfileLocation` structure's `u` field to point to an existing profile.

```
struct CMPtrLocation {
    Ptr  p;
};
```

Field description

`p` A data structure of type `Ptr` holding a pointer that points to the location of a profile in memory. For a description of the `Ptr` data structure, see *Inside Macintosh: Memory.*

Apple Profile Header

Your application cannot obtain a discrete profile header value using the element tag scheme available for use with elements outside the header. Instead, to set or modify values of a profile header, your application must obtain the entire profile header using the `CMGetProfileHeader` function described on page 3-64 and replace the modified header using the `CMSetProfileHeader` function described on page 3-72. To obtain and replace the header for either profile version, these functions take a union of type `CMAppleProfileHeader` with variants for ColorSync 1.0 profile headers and ColorSync Manager version 2.0 profile headers.

```
union CMAppleProfileHeader {
    CMHeader        cm1;
    CM2Header       cm2;
};
```

Field descriptions

`cm1` A version 1.0 profile header. For a description of the ColorSync version 1.0 profile header, see "Profile Header for ColorSync 1.0" on page 3-45.

`cm2` A version 2.0 profile header. For a description of the ColorSync version 2.0 profile header, see "Profile 2.0 Header Structure for the ColorSync Manager" on page 3-26.

Profile 2.0 Header Structure for the ColorSync Manager

To set or modify elements within a ColorSync Manager version 2.0 profile header, your application must obtain the entire profile header using the

CMGetProfileHeader function described on page 3-64 and replace the header using the CMSetProfileHeader function described on page 3-72.

The ColorSync Manager version 2.0 defines the following CM2header profile structure which supports the header format specified by the ICC format specification 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;
    unsigned long           flags;
    OSType                  deviceManufacturer;
    unsigned long           deviceModel;
    unsigned long           deviceAttributes[2];
    unsigned long           renderingIntent;
    CMFixedXYZColor         white;
    char                    reserved[48];
};
```

Field descriptions

size
: The total size in bytes of the profile.

CMMType
: The signature of the preferred CMM to be used for color-matching and color-checking sessions for this profile. To obviate conflicts with other CMMs, this signature must be registered with the ICC. For the signature of the Apple-supplied CMM, see “Signature of the Apple-Supplied Color Management Module” on page 3-9.

profileVersion
: The version of the profile format. The first 8 bits indicate the major version number, followed by 8 bits indicating the minor version number. The following 2 bytes are reserved.

  The profile version number is not tied to the version of the ColorSync Manager. Profile formats and their versions are defined by the ICC. For example, a major version change

may indicate the addition of new required tags to the profile format; a minor version change may indicate the addition of new optional tags.

`profileClass` One of the six types of profile classes supported by the ICC: input, display, output, device link, color space conversion or abstract. For the signatures representing profile classes, see "Profile Classes" on page 3-8.

`dataColorSpace` The color space of the profile. Color values used to express colors of images using this profile are specified in this color space. For a list of the color space signatures, see "Color Space Signatures" on page 3-13.

`profileConnectionSpace`
The profile connection space or PCS. The profile connection spaces are `cmXYZData` or `cmLabData`. For the signatures for these two color spaces, see "Color Space Signatures" on page 3-13.

`dateTime` The date and time when the profile was created. You can use this value to keep track of your own versions of this profile.

`CS2profileSignature`
The `'acsp'` constant as required by the ICC format.

`platform` The signature of the primary platform on which this profile runs. For Apple Computer, this is `'APPL'`. For other platforms, refer to the *International Color Consortium Profile Format Specification.* See "Introduction to the ColorSync Manager" in *Advanced Color Imaging on the Mac OS* for information on how to obtain this document. This value is registered with the ICC.

`flags` Flags to indicate hints for the preferred CMM, such as quality and speed options. The `flags` field consists of an unsigned long data type. The low word of the 16 bits is reserved for use by the ICC. The high word is available for use by color management systems. The ColorSync Manager uses the high word in the following way: it uses the least significant 2 bits of the high word for the quality and speed flag. This flag specifies the quality for color matching, which can be normal mode, draft mode, or best mode. Best mode is slowest, but it produces the highest

quality result. (This feature is provided by the ColorSync Manager; it is not defined by the ICC profile specification.)

Use of the first 2 bits of the low word is prescribed by the ICC. The first two bits are set in the following way:

The first bit at position 0 is used to indicate if the profile is embedded. The bit is set in the following way: `cmEmbeddedProfile = 0` if the profile is embedded and `1` if the profile is not embedded in a file.

The second bit at position 1 is used to indicate if the profile can be used independently: `cmEmbeddedUse = 0` if the profile can be used independently and `1` if it is to be used as an embedded profile only. You should interpret the setting of this bit as an indication of copyright protection. If the profile developer set this bit to 1, you should use this profile as an embedded profile only and not copy the profile for your own purposes. The profile developer also specifies explicit copyright intention using the `cprt` profile tag.

`deviceManufacturer`

The signature of the manufacturer of the device to which this profile applies. This value is registered with the ICC.

`deviceModel` The model of this device, as registered with the ICC.

`deviceAttributes`

Attributes that are unique to this particular device setup, such as media type, paper, and ink types. This field consists of an array of 2 unsigned long data types, [0] and [1]. The low word of long [1] is reserved by the ICC. The high word of long [1] and the entire word of long [0] are available for your use. Each of the first two bits is set to 1 if the flag is on and 0 if it is off. The first bit at position 0 is set to 1 if the media is transparency and 0 if the media is reflective. The second bit at position 1 is set to 1 if the media is matte and 0 if the media is glossy.

`renderingIntent`

The preferred rendering intent for the object or file tagged with this profile. Rendering intents are perceptual, relative colorimetric, saturation, and absolute colorimetric.This field consists of an unsigned long data type. The low word is reserved by the ICC. The high word is available for use.

	The ColorSync Manager uses the high word for setting the rendering intent. You can use the constants defined by the ColorSync Manager for the rendering intents to set this field. See "Rendering Intent Values for Version 2.0 Profiles" on page 3-19. For information about rendering intents, see the chapter "Developing ColorSync-Supportive Applications" in *Advanced Color Imaging on the Mac OS*.
`white`	The profile illuminant white reference point which is expressed in the XYZ color space.
`reserved`	This field is reserved for future use.

Concatenated Profile Set Structure

A color world is not limited to two profiles. It can include a series of profiles that describe the processing to be carried out in a work flow sequence such as scanning, printing, and previewing an image.

To identify a set of profiles, your application uses a data structure of type `CMConcatProfileSet` that includes an array to hold the set of profile references. You provide this array as the `profileSet` field of the `CMConcatProfileSet` structure. You specify the profiles of the array in processing order—from source through destination.

The array identifies a concatenated profile set your application can use to establish a color world in which the sequential relationship among the profiles exists until your application disposes of the color world. Alternatively, you can create a device-linked profile composed of a series of linked profiles that remains intact and available for use again after your application disposes of the concatenated color world. In either case, you use a data structure of type `CMConcatProfileSet` to define the profile set.

A device-linked profile accommodates users who use a specific configuration requiring a combination of device profiles and possibly nondevice profiles repeatedly over time.

To set up a color world that includes a concatenated set of profiles, your application uses the `CWConcatColorWorld` function, passing it a structure of type `CMConcatProfileSet`. For a description of the `CWConcatColorWorld` function, see page 3-82. Your application may use the `CMConcatProfileSet` structure to pass the `CWConcatColorWorld` function an array containing a set of profile references or an array containing only the profile reference of a device-linked profile.

To create a device-linked profile, your application calls the `CWNewLinkProfile` function passing it a structure of type `CMConcatProfileSet`. For a description of the `CWNewLinkProfile` function, see page 3-84.

```
struct CMConcatProfileSet {
    unsigned short      keyIndex;
    unsigned short      count;
    CMProfileRef        profileSet[1];
};
```

Field descriptions

`keyIndex`	A zero-based index into the array of profile references identifying the profile whose CMM is to be used for the entire session. The profile's `CMMType` field identifies the CMM.
`count`	The number of profiles in the profile array. A minimum of one profile is required.
`profileSet`	A variable-length array of profile references. The profiles whose references you specify must be in processing order from source to destination. The rules governing the types of profiles you can specify in a profile array differ depending on whether you are creating a profile set for the `CWConcatColorWorld` function or for the `CWNewLinkProfile` function. See the functions for details.

Color World Information Record

Your application supplies a color world information record structure of type `CMCWInfoRecord` as a parameter to the `CMGetCWInfo` function to obtain information about a given color world. The ColorSync Manager uses this data structure to return information about the color world.

```
struct CMCWInfoRecord {
    unsigned long       cmmCount;
    CMMInfoRecord       cmmInfo[2];
};
```

Field descriptions

`cmmCount` The number of CMMs involved in the color-matching session, either 1 or 2.

`cmmInfo` An array containing two elements. Depending on the value that `cmmCount` returns, the `cmmInfo` array contains one or two records of type `CMMInfoRecord` reporting the CMM type and version number.

If `cmmCount` is 1, the first element of the array (`cmmInfo[0]`) identifies the CMM and the second element of the array (`cmmInfo[1]`) is undefined.

If `cmmCount` is 2, the first element of the array (`cmmInfo[0]`) identifies the source CMM and the second element of the array (`cmmInfo[1]`) identifies the destination CMM. For a description of the `CMMInfoRecord` data structure, see "Color Management Module (CMM) Information Record Structure" on page 3-32.

Color Management Module (CMM) Information Record Structure

Your application supplies an array containing two CMM information record structures of type `CMMInfoRecord` as a field of the `CMCWInfoRecord` structure. These structures allow the `CMGetCWInfo` function to return information about the one or two CMMs used in a given color world. Your application must allocate memory for the array. When your application calls the `CMGetCWInfo` function, it passes a pointer to the `CMCWInfoRecord` structure containing the array.

```
struct CMMInfoRecord {
    OSType      CMMType;
    long        CMMVersion;
};
```

Field descriptions

`CMMType` The signature of the CMM as specified in the profile header's `CMMType` field. The `CMGetCWInfo` function returns this value.

`CMMVersion` The version of the CMM. The `CMGetCWInfo` function returns this value.

Profile Search Record

Your application supplies a search record of type `CMSearchRecord` as the `searchSpec` parameter to the `CMNewProfileSearch` function to provide the ColorSync Manager with the search criteria to use in determining which version 2 profiles to include in the result list and which to filter out. You cannot use the ColorSync Manager search functions to search for ColorSync 1.0 profiles.

The ColorSync Manager preserves this information internally along with the search result list until your application calls the `CMDisposeProfileSearch` function to release the memory. This allows your application to call the `CMUpdateProfileSearch` function to update the search result if the ColorSync™ Profiles folder contents change without needing to provide the search specification again. A search record is defined by the `CMSearchRecord` type definition.

```
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;
};
```

Constant descriptions

`CMMType` The signature of a CMM. The `CMMType` field of a profile's header must specify this signature if the `searchMask` bitmask you specify includes this field.

`profileClass` The class of profile to search for. The `profileClass` field of a profile's header must specify this signature if the `searchMask` bitmask you specify includes this field. For a list of profile classes and the constants for their signatures, see "Profile Classes" on page 3-8.

`dataColorSpace` A data color space. The `dataColorSpace` field of a profile's header must specify this value if the `searchMask` bitmask

you specify includes this field. For a list of the color space signatures, see "Color Space Signatures" on page 3-13.

`profileConnectionSpace`
: A profile connection color space. The `profileConnectionSpace` field of a profile's header must match this value if the `searchMask` bitmask you specify includes this field. The profile connection spaces are `cmXYZData` or `cmLabData`. For the signatures of these two color spaces, see "Color Space Signatures" on page 3-13.

`deviceManufacturer`
: The signature of the manufacturer. The `deviceManufacturer` field of a profile's header must match this value if the `searchMask` bitmask you specify includes this field.

`deviceModel`
: The model of a device. The `deviceModel` field of a profile's header must match this value if the `searchMask` bitmask you specify includes this field.

`deviceAttributes`
: Attributes for a particular device setup, such as media type, paper, and ink types. The `deviceAttributes` field of a profile's header must match these attributes if the `searchMask` bitmask you specify includes this field.

`profileFlags`
: Flags that indicate hints for the preferred CMM, such as quality, speed, and memory options. The `flags` field of a profile's header must specify this value if the `searchMask` bitmask you specify includes this field. In most cases, you will not want to search for profiles based on the flags settings.

`searchMask`
: A bitmask that specifies the search record fields to be used in the profile search. Here are the defined bitmask values:

cmMatchAnyProfile	0x00000000
cmMatchProfileCMMType	0x00000001
cmMatchProfileClass	0x00000002
cmMatchDataColorSpace	0x00000004
cmMatchProfileConnectionSpace	0x00000008
cmMatchManufacturer	0x00000010
cmMatchModel	0x00000020
cmMatchAttributes	0x00000040
cmMatchProfileFlags	0x00000080

`filter` A pointer to a calling program-supplied function. This function examines a profile to determine if it should be excluded from the profile search result list based on criteria such as an element or elements not included in the search record fields. For more information, see the `MyCMProfileFilterProc` function on page 3-136.

XYZ Color Component Values

Three components combine to express a color value defined by the `CMXYZColor` type definition in the XYZ color space. Each color component is described by a numeric value defined by the `CMXYZComponent` type definition. A component value of type `CMXYZComponent` is expressed as a 16-bit value. This is formatted as an unsigned value with 1 bit of integer portion and 15 bits of fractional portion.

```
typedef unsigned short CMXYZComponent;
```

XYZ Color Value

Color component values defined by `CMXYZComponent` type definition combine to form a color value specified in the XYZ color space. The color value is defined by the `CMXYZColor` type definition.

Your application uses the `CMXYZColor` data structure to specify a color value in the `CMColor` union to be used in low-level color matching, color checking, or color conversion. You also use the `CMXYZColor` data structure to specify the XYZ white point reference used in the conversion of colors to or from the XYZ color space.

```
struct CMXYZColor {
    CMXYZComponent      X;
    CMXYZComponent      Y;
    CMXYZComponent      Z;
};
```

Fixed XYZ Color Value

The `CMFixedXYZColor` data type is used to specify the profile illuminant in the profile header's `white` field and to specify other profile element values. Your

application uses the CMFixedXYZColor data type to specify color values to be converted to and from color values defined by the CMXYZColor data type.

Color component values defined by the Fixed type definition can be used to specify a color value in the XYZ color space with greater precision than a color whose components are expressed as CMXYZComponent data types. The Fixed data type is a signed 32-bit value. A color value expressed in the XYZ color space whose color components are of type Fixed is defined by the CMFixedXYZColor type definition.

To convert color values, you use the CMFixedXYZToXYZ function described on page 3-117 and the CMXYZToFixedXYZ function described on page 3-116.

```
struct CMFixedXYZColor {
    Fixed           X;
    Fixed           Y;
    Fixed           Z;
};
```

L*a*b* Color Value

A color expressed in the L*a*b* color space is composed of L, a, and b component values. Each color component is expressed as a numeric value within the range of 0 to 65280. For the L component, this maps to 0 to 100 inclusive. For the a component, this maps to –128 to 127 inclusive. For the b component, this maps to –128 to 127 inclusive. The color value is defined by the CMLabColor type definition.

```
struct CMLabColor {
    unsigned short          L;
    unsigned short          a;
    unsigned short          b;
};
```

L*u*v* Color Value

A color value expressed in the L*u*v* color space is composed of L, u, and v component values. Each color component is expressed as a numeric value within the range of 0 to 65535. For the L component, this maps to 0 to 100 inclusive. For the u component, this maps to –128 to 127.996 inclusive. For the

v component, this maps to –128 to 127.996 inclusive. The color value is defined by the CMLuvColor type definition.

```
struct CMLuvColor {
    unsigned short          L;
    unsigned short          u;
    unsigned short          v;
};
```

Yxy Color Value

A color value expressed in the Yxy color space is composed of capY, x, and y component values. Each color component is expressed as a numeric value within the range of 0 to 65535 which maps to 0 to 1. The color value is defined by the CMYxyColor type definition

```
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

A color value expressed in the RGB color space is composed of red, green, and blue component values. Each color component is expressed as a numeric value within the range of 0 to 65535.

```
struct CMRGBColor {
    unsigned short          red;
    unsigned short          green;
    unsigned short          blue;
};
```

HLS Color Value

A color value expressed in the HLS color space is composed of hue, lightness, and saturation component values. Each color component is expressed as a

numeric value within the range of 0 to 65535 inclusive. The `hue` value represents a fraction of a circle in which red is positioned at 0.

```
struct CMHLSColor {
    unsigned short          hue;
    unsigned short          lightness;
    unsigned short          saturation;
};
```

HSV Color Value

A color value expressed in the HSV color space is composed of `hue`, `saturation`, and `value` component values. Each color component is expressed as a numeric value within the range of 0 to 65535 inclusive. The `hue` value represents a fraction of a circle in which red is positioned at 0.

```
typedef struct CMHSVColor {
    unsigned short          hue;
    unsigned short          saturation;
    unsigned short          value;
}CMHSVColor;
```

CMYK Color Value

A color value expressed in the CMYK color space is composed of `cyan`, `magenta`, `yellow`, and `black` component values. Each color component is expressed as a numeric value within the range of 0 to 65535 inclusive.

```
struct CMCMYKColor {
    unsigned short          cyan;
    unsigned short          magenta;
    unsigned short          yellow;
    unsigned short          black;
};
```

CMY Color Value

A color value expressed in the CMY color space is composed of `cyan`, `magenta`, and `yellow` component values. Each color component is expressed as a numeric value within the range of 0 to 65535 inclusive.

```
struct CMCMYColor {
    unsigned short          cyan;
    unsigned short          magenta;
    unsigned short          yellow;
};
```

HiFi Color Values

A color expressed in one of the multichannel color spaces with 5, 6, 7, or 8 channels. The color value for each channel component is expressed as an unsigned byte of type `char`.

```
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

A color value expressed in the Gray color space is composed of a single component, `gray`, represented as a numeric value within the range of 0 to 65535 inclusive.

```
struct CMGrayColor {
    unsigned short      gray;
};
```

The Color Union

Your application may use a union of type `CMColor` to specify a color value defined by one of the 14 data types supported by the union. Your application specifies an array of unions of type `CMColor` containing a list of colors to be matched, checked, or converted. The array is passed as a parameter to the low-level color matching, color checking, or color conversion functions. The following functions use a color union:

- The `CWMatchColors` function, described on page 3-97, matches the colors in the color list array to the data color space of the destination profile specified by the color world.
- The `CWCheckColors` function, described on page 3-98, checks the colors in the color list array against the color gamut specified by the color world's destination profile.
- The color conversion functions, described from page 3-108 to page 3-114, take source and destination array parameters of type `CMColor` specifying lists of colors to be converted from one color space to another.

You do not use a union of type `CMColor` to convert colors expressed in the XYZ color space as values of type `CMFixedXYZ` because the `CMColor` union does not support the `CMFixedXYZ` data type.

The color union is defined by the `CMColor` type definition.

```
union CMColor {
    CMRGBColor              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;
    };
```

A color union can contain one of the following fields.

Field descriptions

`rgb` A color value expressed in the RGB color space as data of type `CMRGBColor`. See "RGB Color Value" on page 3-37 for a description of the `CMRGBColor` data type.

`hsv` A color value expressed in the HSV color space as data of type `CMHSVColor`. See "HSV Color Value" on page 3-38 for a description of the `CMHSVColor` data type.

`hls` A color value expressed in the HLS color space as data of type `CMHLSColor`. See "HLS Color Value" on page 3-37 for a description of the `CMHLSColor` data type.

`XYZ` A color value expressed in the XYZ color space as data of type `CMXYZColor`. See "XYZ Color Value" on page 3-35 for a description of the `CMXYZColor` data type.

`Lab` A color value expressed in the L*a*b* color space as data of type `CMLabColor`. See "L*a*b* Color Value" on page 3-36 for a description of the `CMLabColor` data type.

`Luv` A color value expressed in the L*u*v* color space as data of type `CMLuvColor`. See "L*u*v* Color Value" on page 3-36 for a description of the `CMLuvColor` data type.

`Yxy` A color value expressed in the Yxy color space as data of type `CMYxyColor`. See "Yxy Color Value" on page 3-37 for a description of the `CMYxyColor` data type.

`cmyk` A color value expressed in the CMYK color space as data of type `CMCMYKColor`. See "CMYK Color Value" on page 3-38 for a description of the `CMCMYKColor` data type.

`cmy` A color value expressed in the CMY color space as data of type `CMCMYColor`. See "CMY Color Value" on page 3-39 for a description of the `CMCMYColor` data type.

`gray` A color value expressed in the Gray color space as data of type `CMGrayColor`. See "Gray Color Value" on page 3-39 for a description of the `CMGrayColor` data type.

`mc5` A color value expressed in the five-channel multichannel color space as data of type `CMMultichannel5Color`. See “HiFi Color Values” on page 3-39 for a description of the `CMMultichannel5Color` data type.

`mc6` A color value expressed in the six-channel multichannel color space as data of type `CMMultichannel6Color`. See “HiFi Color Values” on page 3-39 for a description of the `CMMultichannel6Color` data type.

`mc7` A color value expressed in the seven-channel multichannel color space as data of type `CMMultichannel7Color`. See “HiFi Color Values” on page 3-39 for a description of the `CMMultichannel7Color` data type.

`mc8` A color value expressed in the eight-channel multichannel color space as data of type `CMMultichannel8Color`. See “HiFi Color Values” on page 3-39 for a description of the `CMMultichannel8Color` data type.

The ColorSync Manager Bitmap

The ColorSync Manager provides a bitmap structure of type `CMBitmap` to describe color bitmap images. When your application calls the `CWMatchBitmap` function, described on page 3-92, you pass a pointer to the source bitmap of type `CMBitMap` containing the image whose colors are to be matched to the color gamut of the destination device’s image specified by the destination profile of the given color world. If you do not want the image color matched in place, you can also pass a pointer to a resulting bitmap of type `CMBitmap` to define and hold the color matched image. When your application calls the `CWCheckBitMap` function, described on page 3-95, it passes a pointer to the source bitmap of type `CMBitMap` describing the source image and a pointer to a resulting bitmap of type `CMBitMap` to hold the color check results.

IMPORTANT

For QuickDraw GX, an image can have an indexed bitmap to a list of colors. The ColorSync Manager does not support indexed bitmaps. Instead, your application can use the low-level matching functions to match the individual elements of the color table. ▲

```
struct CMBitmap {
    char                    *image;
    long                    width;
    long                    height;
    long                    rowBytes;
    long                    pixelSize;
    CMBitmapColorSpace      space;
    long                    user1;
    long                    user2;
    };
```

Field descriptions

`image` A pointer to a bit image.

`width` The width of the bit image, that is, the number of pixels in a row.

`height` The height of the bit image, that is, the number of rows in the image.

`rowBytes` The offset in bytes from one row of the image to the next.

`pixelSize` The number of bits per pixel.

`space` The color space in which the colors of the bitmap image are specified. For a description of the possible color spaces for color bitmaps, see "Color Spaces," beginning on page 3-15.

`user1` Not used by ColorSync. This field is reserved for use by QuickDraw GX.

`user2` Not used by ColorSync. This field is reserved for use by QuickDraw GX.

Profile Reference

A profile reference is the means by which your application gains access to a profile. Several ColorSync Manager functions return a profile reference to your application. Your application then passes it as a parameter on subsequent calls to other ColorSync Manager functions that use profiles.

The ColorSync Manager returns a unique profile reference in response to each individual call to the `CMOpenProfile`, `CMCopyProfile`, and `CMNewProfile` functions described beginning on page 3-50. This allows multiple applications concurrent

access to a profile. The ColorSync Manager defines an abstract private data structure of type `CMPrivateProfileRecord` for the profile reference.

```
typedef struct CMPrivateProfileRecord *CMProfileRef;
```

Profile Search Result Reference

A search result consists of a list of profiles matching certain search criteria. When your application calls the `CMNewProfileSearch` function described on page 3-101 to search in the ColorSync Profiles folder for profiles that meet certain criteria, the ColorSync Manager returns a reference to an internal private data structure containing the search result. In subsequent calls to the ColorSync Manager functions, your application passes the search result reference to the function to update the search result list, dispose of it, open a reference to a profile at a specific position in the list, or to obtain the file specification for a profile in the list. The ColorSync Manager defines an abstract private data structure of type `CMPrivateProfileSearchResult` for the search result reference.

```
struct CMPrivateProfileSearchResult *CMProfileSearchRef;
```

High-Level Color-Matching-Session Reference

The ColorSync Manager defines an abstract private data structure of type `CMPrivateMatchRefRecord` for the color-matching-session reference. When your application calls the `NCMBeginMatching` function described on page 3-75 to begin a high-level color-matching session, the ColorSync Manager returns a reference to the color-matching session which you must later pass to the `CMEndMatching` function to conclude the session.

```
struct CMPrivateMatchRefRecord *CMMatchRef;
```

Color World Reference

Your application passes a color world reference as a parameter on calls to functions to hold color-matching and color-checking sessions and to dispose of the color world. When your application calls the `NCWNewColorWorld` function described on page 3-81 and the `CWConcatColorWorld` function described on page 3-82 to allocate a color world for color-matching and color-checking

sessions, the ColorSync Manager returns a reference to the color world. The ColorSync Manager defines an abstract private data structure of type `CMPrivateColorWorldRecord` for the color world reference.

```
struct CMPrivateColorWorldRecord *CMWorldRef;
```

TEnableColorMatchingBlk

You pass a structure defined by the `TEnableColorMatchingBlk` data type to the `PrGeneral` function when you use the `EnableColorMatchingOp` opcode, described in "Operation Codes Used With PrGeneral Function" on page 3-22. ColorSync-supportive drivers support the `EnableColorMatchingOp` operation code as a `PrGeneral` call that turns the `fEnableIt` flag on or off to enable or disable color matching.

```
struct TEnableColorMatchingBlk {
    short                       iOpCode;
    short                       iError;
    long                        lReserved;
    THPrint                     hPrint;
    Boolean                     fEnableIt;
    SInt8                       filler;
    };
```

Field descriptions

`iOpCode`	The `PrGeneral` printing opcode.
`iError`	The returned error code.
`lReserved`	Reserved for future use.
`hPrint`	A valid print record.
`fEnableIt`	The flag set by the `EnableColorMatchingOp` opcode.
`SInt8`	Filler.

Profile Header for ColorSync 1.0

ColorSync 1.0 defines a version 1.0 profile whose structure and format are different from that of the ICC version 2.0 profile.

Your application cannot use the ColorSync Manager API to update a version 1.0 profile or to search for version 1.0 profiles. However, your application can

use the remaining ColorSync Manager API that pertains to profiles with version 1.0 profiles.

Your application can open a version 1.0 profile using the `CMOpenProfileFile` function, obtain the version 1.0 profile header using the `CMGetProfileHeader` function, and access version 1.0 profile elements using the `CMGetProfileElement` function. To make this possible, the ColorSync Manager API includes a union, described in "Apple Profile Header" on page 3-26, that supports either profile header version. The `CMHeader` data type defines the version 1.0 profile header.

```
struct CMHeader {
    unsigned long       size;
    OSType              CMMType;
    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;
};
```

Field descriptions

`size` — The total size in bytes of the profile, including any custom data.

`CMMType` — The signature of the preferred CMM to be used for color-matching and color-checking sessions for this profile. To obviate conflicts with other CMMs, this signature must be registered with the ICC. For the signature of the Apple-supplied CMM, see "Signature of the Apple-Supplied Color Management Module" on page 3-9.

`applProfileVersion` — The Apple profile version. Set this field to $0100 (defined as the constant `kCMApplProfileVersion`).

`dataType` — The kind of color data. The types are

```
rgbData = 'RGB ',  source or destination profiles
cmykData = 'CMYK', destination profiles
grayData = 'GRAY', source or destination profiles
xyzData = 'XYZ '   source or destination profiles
```

`deviceType` The kind of device. The types are

```
monitorDevice = 'mntr'
scannerDevice = 'scnr'
printerDevice = 'prtr'
```

`deviceManufacturer` A name supplied by the device manufacturer.

`deviceModel` The device model specified by the manufacturer.

`deviceAttributes` Private information such as paper surface and ink temperature.

`profileNameOffset` The offset to the profile name from the top of data.

`customDataOffset` The offset to any custom data from the top of data.

`flags` A field used by drivers; it can hold one of the following:

```
CMNativeMatchingPreferred
CMTurnOffCache
```

The `CMNativeMatchingPreferred` flag is available for developers of intelligent peripherals that can off-load color matching into the peripheral. Most drivers will not use this flag. (Its default setting is 0, meaning that the profile creator does not care whether matching occurs on the host or the device.)

The `CMTurnOffCache` flag can be used by CMMs that won't benefit from a cache, such as those that can look up data from a table with less overhead, or that don't want to take the memory hit a cache entails, or that do their own caching and don't want the CMM to do it. (The default is 0, meaning turn on cache.)

`options` The `options` field specifies the kind of matching this profile is for; it can be `CMPerceptualMatch`, the default, `CMColorimetricMatch`, or `CMSaturationMatch`. The options are set by the image creator.

`white` The white point for this profile expressed in XYZ space.

`black` The black point for this profile expressed in XYZ space.

PostScript Color Rendering Dictionary (CRD) Virtual Memory Size Tag Structure

To specify the maximum virtual memory size of the color rendering dictionary (CRD) for a specific rendering intent for a particular PostScript Level 2 printer type, a printer profile can include the Apple-defined `'psvm'` optional tag. This tag's element data includes an array containing one entry for each rendering intent and its virtual memory size.

If a PostScript printer profile includes this tag, the Apple-supplied CMM will use the tag and return the values specified by the tag when your application or device driver calls the `CMGetPS2ColorRenderingVMSize` function described on page 3-129.

If a PostScript printer profile does not include this tag, the CMM uses an algorithm to determine the VM size of the CRD, which may be assessed as greater than the actual maximum VM size.

The `CMIntentCRDVMSize` data type defines the rendering intent and its maximum VM size. The `CMPS2CRDVMSizeType` data type for the tag includes an array containing one or more members of type `CMIntentCRDVMSize`.

```
struct CMIntentCRDVMSize {
    long                renderingIntent;
    unsigned long       VMSize;
};
```

For example, a rendering intent might be 0 and its VM size 120 KB.

Constant descriptions

`renderingIntent` — The rendering intent whose CRD virtual memory size you want to obtain. Rendering intent values are

- `0` (`cmPerceptual`)
- `1` (`cmRelativeColorimetric`)
- `2` (`cmSaturation`)
- `3` (`cmAbsoluteColorimetric`)

`VMSize` — The virtual memory size of the CRD for the rendering intent specified for the `renderingIntent` field.

The `CMPS2CRDVMSizeType` data type defines the Apple-defined `'psvm'` optional tag.

```
struct CMPS2CRDVMSizeType {
    OSType              typeDescriptor;
    unsigned long       reserved;
    unsigned long       count;
    CMIntentCRDVMSize   intentCRD[1];
};
```

Constant descriptions

`typeDescriptor` The `'psvm'` tag signature.

`reserved` Reserved for future use.

`count` The number of entries in the `intentCRD` array. You should specify at least 4 entries: 0, 1, 2, and 3.

`CMIntentCRDVMSize`
A variable-sized array of four or more members defined by the `CMIntentCRDSize` data type.

The ColorSync Manager Functions

This section describes the functions defined for your application's use by the ColorSync Manager.

The functions are organized into the following categories:

- profile file and element access
- high-level QuickDraw-specific matching
- low-level matching
- system profile access
- external profile searching
- color conversion
- PostScript-support functions
- utilities
- calling-program-supplied function prototypes

Accessing Profile Files

This section describes the functions you use to open, update, close, create, copy, validate, flatten, and unflatten profiles.

CMOpenProfile

To open a specific profile and receive a reference to the profile, use the `CMOpenProfile` function.

```
pascal CMError CMOpenProfile (CMProfileRef *prof
                              const CMProfileLocation *theProfile);
```

`prof` A reference to a unique internal private data structure. For more information, see "Profile Reference" on page 3-43.

`theProfile` The location of the profile, which you specify using the `CMProfileLocation` data type described on page 3-24. Commonly a profile is disk-file based. However, the profile may be a file-based profile, a handle-based profile, or a pointer-based profile.

DESCRIPTION

If the `CMOpenProfile` function completes successfully, the profile reference is returned to your application. You need this reference to identify the profile to be used when your application calls functions, for example, to color match, copy, and update a profile, and validate its contents.

The ColorSync Manager maintains private storage for each request to open a profile, allowing more than one application to use a profile concurrently.

When you create a new profile or modify the elements of an existing profile, the ColorSync Manager stores the new or modified elements in the private storage it maintains for your application. Any new or changed profile elements are not incorporated into the profile itself until your application calls the `CMUpdateProfile` function, described on page 3-52, to update the profile. If you call the `CMCopyProfile` function, described on page 3-55, to create a copy of an existing profile under a new name, any changes you have made are

incorporated in the profile duplicate but the original profile remains unchanged.

Before you call the `CMOpenProfile` function, you must set the `CMProfileLocation` data structure to identify the location of the profile to be opened. Most commonly, a profile is stored in a disk file. If the profile is in a disk file, use the profile location data type to provide its file specification. If the profile is in memory, use the profile location data type to specify a handle or pointer to the profile.

Your application must obtain a profile reference before you copy or validate a profile, and before you flatten the profile to embed it.

For example, your application may open a profile, call the `CMGetProfileHeader` function to obtain the profile's header in order to modify its values, set new values, call the `CMSetProfileHeader` function to replace the modified header, and finally pass the profile reference to a function such as `NCWNewColorWorld` to be used as the source or destination profile in a color world for a color-matching session.

When you close your reference to the profile by calling the `CMCloseProfile` function, described on page 3-51, your changes are discarded.

CMCloseProfile

To close a reference to a profile, use the `CMCloseProfile` function.

```
pascal CMError CMCloseProfile (CMProfileRef prof);
```

`prof` The profile reference to be closed. For more information, see "Profile Reference" on page 3-43.

DESCRIPTION

The `CMCloseProfile` function closes the specified reference to a profile returned from a previous call to the `CMOpenProfile` or `CMNewProfile` functions.

The `CMCloseProfile` function releases memory allocated in association with the profile reference. Any temporary changes your application made to the profile are not recorded in the profile before the profile reference is closed unless you first call the `CMUpdateProfile` function to update the profile.

The `CMCloseProfile` function does not close the profile itself. Instead, it closes your application's unique reference to the profile. The profile will remain open if other references to it exist. The ColorSync Manager closes the profile when the last reference to the profile is closed.

If you create a new profile by calling the `CMNewProfile` function, the profile is saved to disk when you call the `CMCloseProfile` function unless you specified `NULL` as the profile location when you created the profile.

SEE ALSO

To save changes to a profile before closing it, use the `CMUpdateProfile` function, which is described next.

CMUpdateProfile

To save modifications to a profile, use the `CMUpdateProfile` function.

```
pascal CMError CMUpdateProfile (CMProfileRef prof);
```

`prof` A reference to the profile to be updated. For more information, see "Profile Reference" on page 3-43.

DESCRIPTION

The `CMUpdateProfile` function makes permanent any changes or additions your application has made to the profile indicated by the profile reference if no other references to that profile exist.

Each time an application calls the `CMOpenProfile` function, a unique reference to the profile is created. More than one reference to a profile may exist. If the profile is opened by another program when your application calls this function, the ColorSync Manager returns an error and does not update the profile.

You cannot use the `CMUpdateProfile` function to update a ColorSync 1.0 profile. For information on updating a ColorSync 1.0 profile, see the appendix, "ColorSync Manager Backward Compatibility"

SEE ALSO

After you fill in tags and their data elements for a new profile created by calling the `CMNewProfile` function, described on page 3-53, you must call `CMUpdateProfile` to write the element data to the new profile.

If you modify an open profile, you must call `CMUpdateProfile` to save the changes to the profile file before you call `CMCloseProfile`, described on page 3-51. Otherwise, the changes are discarded.

To modify a profile header, you use the `CMGetProfileHeader` function described on page 3-64 and the `CMSetProfileHeader` function described on page 3-72.

To set profile elements outside the header, you use the `CMSetProfileElement` function described on page 3-71, the `CMSetProfileElementSize` function described on page 3-69, and the `CMSetPartialProfileElement` function described on page 3-70.

CMNewProfile

To create a new profile and associated backing copy, use the `CMNewProfile` function.

```
pascal CMError CMNewProfile (CMProfileRef *prof,
                             const CMProfileLocation *theProfile);
```

`prof` A reference to the profile that the ColorSync Manager returns if the function completes successfully. For more information, see "Profile Reference" on page 3-43.

`theProfile`
The location for the new profile. You use the `CMProfileLocation` data type, described on page 3-24, to specify the profile location. The default disk file type for a profile is `prof`. A profile is commonly disk-file based. However, to accommodate special requirements, you can create a new profile in relocatable memory that is a handle-based profile or you can create a temporary profile that isn't saved after you call the `CMCloseProfile` function. To create a temporary profile, you can

either specify `cmNoProfileBase` as the kind of profile in the profile location structure or you can specify `NULL` for this parameter.

DESCRIPTION

The `CMNewProfile` function creates a new profile and backing copy in the location you specify. After you create the profile, you must fill in the profile header fields and populate the profile with tags and their element data, and then call `CMUpdateProfile`, described on page 3-52, to save the element data to the profile file. The default ColorSync 2.0 profile contents include a profile header of type `CM2Header`, described on page 3-26, and an element table.

To set profile elements outside the header, you use the `CMSetProfileElement` function described on page 3-71, the `CMSetProfileElementSize` function described on page 3-69, and the `CMSetPartialProfileElement` function described on page 3-70. You set these elements individually, identifying them by their tag names.

When you create a new profile, all fields of the `CM2Header` profile header are set to zero except the `size` and `profileVersion` fields. To set the header elements, you call the `CMGetProfileHeader` function described on page 3-64 to get a copy of the header, assign values to the header fields, then call the `CMSetProfileHeader` function described on page 3-72 to write the new header to the profile.

For each profile type, such as a device profile, there is a specific set of elements and associated tags defined by the ICC that a profile must contain to meet the baseline requirements. The ICC also defines optional tags that a particular CMM might use to optimize or improve its processing. You can also define private tags, whose tag signatures you register with the ICC, to provide a CMM with greater capability to refine its processing.

After you fill in the profile with tags and their element data, you must call the `CMUpdateProfile` function to write the new profile elements to the profile file.

Special Considerations

This function is most commonly used by profile developers who create profiles for device manufacturers and by calibration applications. In most cases, application developers use existing profiles. ◆

SEE ALSO

For information on how to fill in a profile with tags and element data including a description of the profile tags, refer to the *International Color Consortium Profile Format Specification*. See *Advanced Color Imaging on the Mac OS*, "Introduction to the ColorSync Manager" for information on how to obtain this document.

CMCopyProfile

To duplicate an existing profile, use the `CMCopyProfile` function.

```
pascal CMError CMCopyProfile (CMProfileRef *targetProf,
                        const CMProfileLocation *targetLocation,
                        CMProfileRef srcProf);
```

`targetProf` A reference to the copy of the profile. The ColorSync Manager returns this reference to your application if the function completes successfully. For more information, see "Profile Reference" on page 3-43.

`targetLocation`
The location in memory or on disk where the ColorSync Manager is to create the copy of the profile. A profile is commonly disk-file based. However, to accommodate special requirements, you can create a new profile in relocatable memory that is a handle-based profile or you can create a temporary profile that isn't saved after you call the `CMCloseProfile` function. To create a temporary profile, you can either specify `cmNoProfileBase` as the kind of profile in the profile location structure or you can specify `NULL` for this parameter. To specify the location, you use the `CMProfileLocation` data type described on page 3-24.

`srcProf` The reference for the profile to be duplicated.

DESCRIPTION

The `CMCopyProfile` function duplicates an existing open profile whose reference you specify. If you have made temporary changes to the profile, which you have not saved by calling `CMUpdateProfile`, those changes are included in the

copy of the profile to be created. They are not saved to the original profile unless you call `CMUpdateProfile` for that profile.

Unless you are copying a profile that you created, you should not infringe on copyright protection specified by the profile creator. To obtain the copyright information, you call the `CMGetProfileElement` function, described on page 3-62, specifying the `cprt` tag signature for the copyright element. You should also check the `flags` field of the `CMProfileHeader`, described in "Profile 2.0 Header Structure for the ColorSync Manager," beginning on page 3-26, for copyright information. The second bit of the flags field at position 1 is used to indicate if the profile can be used independently. If the profile developer set this bit to 1, you should use this profile as an embedded profile only and not copy the profile for your own purposes.

A calibration program might use this function, for example, to copy a device's original profile, then modify the copy to reflect the current state of the device. You might also want to copy a profile after you unflatten it.

SEE ALSO

To copy a profile, you must obtain a reference to that profile by either opening the profile or creating it. To open a profile, use the `CMOpenProfile` function described on page 3-50. To create a new profile, use the `CMNewProfile` function described on page 3-53.

CMGetProfileLocation

To identify the physical file location of a profile given a profile reference, use the `CMGetProfileLocation` function.

```
pascal CMError CMGetProfileLocation(CMProfileRef prof,
                          CMProfileLocation *theProfile)
```

`prof` A reference to the profile to be tested. For information on profile references, see "Profile Reference" on page 3-43.

`theProfile` The physical location of the profile. This value is returned if the function completes successfully. See "Profile Location Structure" on page 3-24.

SEE ALSO

To open a profile and obtain a reference to it, use the `CMOpenProfile` function described on page 3-50.

CMValidateProfile

To test if a profile contains the minimum set of elements required by the CMM to be used for color matching or color checking, use the `CMValidateProfile` function.

```
pascal CMError CMValidateProfile (CMProfileRef prof,
                        Boolean *valid,
                        Boolean *preferredCMMnotfound);
```

`prof` A reference to the profile to be tested. For information on profile references, see "Profile Reference" on page 3-43.

`valid` A flag that returns `true` if the profile contains the minimum set of elements and `false` if it doesn't.

`preferredCMMnotfound`
A flag that returns `true` if the CMM specified by the profile was not available to perform the validation or does not support this function and the default Apple-supplied CMM was used. If the profile's CMM is able to perform the test, this flag returns `false`.

DESCRIPTION

When your application calls the `CMValidateProfile` function, the ColorSync Manager dispatches the function to the CMM specified by the `CMMType` header field of the profile whose reference you specify. The preferred CMM can support this function or not.

If the preferred CMM supports this function, it determines if the profile contains the baseline elements for the profile type, which the CMM requires to perform color matching or gamut checking. For each profile type, such as a device profile, there is a specific set of required tagged elements defined by the ICC that the profile must include. The ICC also defines optional tags, which may be included in a profile. A CMM might use these optional elements to optimize or improve its processing. Additionally, a profile might include

private tags defined to provide a CMM with processing capability particular to the needs of that CMM. The profile developer can define these private tags, register the tag signatures with the ICC, and include the tags in a profile. The CMM checks only for the existence of profile elements; it does not check the element's content and size.

If the preferred CMM does not support the `CMValidateProfile` function request, the ColorSync Manager calls the Apple-supplied default CMM to handle the validation request.

CMFlattenProfile

To transfer a profile stored in an independent disk file to an external profile format that can be embedded in a graphics document, use the `CMFlattenProfile` function.

```
pascal CMError CMFlattenProfile (CMProfileRef prof,
                        unsigned long flags,
                        CMFlattenUPP proc, void *refCon,
                        Boolean *preferredCMMnotfound);
```

`prof` A reference to the profile to be flattened. For more information, see "Profile Reference" on page 3-43.

`flags` Reserved for future use.

`proc` A pointer to a function that you provide to perform the low-level data transfer. For a description of the `MyColorSyncDataTransfer` function, see page 3-131.

`refCon` A reference constant for application data which the Color Management Module (CMM) passes to the `MyColorSyncDataTransfer` function each time it calls the function. For example, the reference constant may point to a data structure that holds information required by the `MyColorSyncDataTransfer` function to perform the data transfer, such as the reference number to a disk file in which the flattened profile is to be stored.

`preferredCMMnotfound`
A flag that is set to `true` if the CMM specified by the profile was not available or it does not support this function and the default Apple-supplied CMM was used to flatten the profile. If the profile's CMM supports this function, this flag is set to `false`.

DESCRIPTION

The ColorSync Manager dispatches the `CMFlattenProfile` function to the CMM specified by the profile whose reference you provide. If the preferred CMM is unavailable or it doesn't support this function, then the default Apple-supplied CMM is used.

The ColorSync Manager passes to the CMM the pointer to your profile-flattening function. The CMM calls your `MyColorSyncDataTransfer` function to perform the actual data transfer. For a description of the `MyColorSyncDataTransfer` function declaration, see page 3-131.

SEE ALSO

To unflatten a profile embedded in a graphics document to an independent disk file, use the `CMUnflattenProfile` function, described on page 3-59.

CMUnflattenProfile

To transfer a profile that was embedded in a graphics document to an independent disk file, use the `CMUnflattenProfile` function.

```
pascal CMError CMUnflattenProfile (FSSpec *resultFileSpec,
                        CMFlattenUPP proc, void *refCon,
                        Boolean *preferredCMMnotfound);
```

`resultFileSpec`
The profile file specification for the independent disk file, which is returned if the function completes successfully.

`proc` A pointer to a calling-program-supplied function that you provide to receive the profile data from the CMM and store in a file. For a description of the `MyColorSyncDataTransfer` function, see page 3-131.

`refCon` A reference constant for application data which the CMM passes to the `MyColorSyncDataTransfer` function each time it calls the function.

`preferredCMMnotfound`
A flag that is set to `true` if the CMM specified by the profile was not available or does not support this function and the default Apple-supplied CMM was used to unflatten the profile. If the profile's CMM supports this function, this flag is set to `false`.

DESCRIPTION

The ColorSync Manager dispatches the `CMUnflattenProfile` function to the CMM specified by the profile to be transferred to a disk file. If the preferred CMM is unavailable or it doesn't support this function, then the default Apple-supplied CMM is used.

The ColorSync Manager calls your unflattening function to identify the CMM to which it dispatches the `CMUnflattenProfile` function. For this reason, your function must buffer at least 8 bytes of data. For a description of the `MyColorSyncDataTransfer` unflattening function prototype, see page 3-131.

The CMM calls your `MyColorSyncDataTransfer` function to transfer the profile data from the graphics document to an independent disk file.

Before you can obtain a profile reference to a profile that was embedded in a graphics document, you must use this function to unflatten the profile. Then you can call `CMOpenProfile` to open the profile and obtain a reference to it.

After you're finished with the profile, you must call the `CMCloseProfile` function to close the profile and call the File Manager's `FSpDelete` function to delete the file.

Accessing Profile Elements

This section describes the functions you use to examine, set, and change individual elements of a profile.

CMProfileElementExists

To test whether a given profile contains a specific element based on the element's tag signature, use the `CMProfileElementExists` function.

```
pascal CMError CMProfileElementExists (CMProfileRef prof,
                      OSType tag,
                      Boolean *found);
```

`prof` A reference to the profile. For information on profile references, see "Profile Reference" on page 3-43.

`tag` The tag signature for the element in question. For a complete list of the tag signatures a profile may contain, including a description of each tag, refer to the *International Color Consortium Profile Format Specification.* For information on how to obtain this document, see "Profiles" in the chapter "Introduction to the ColorSync Manager" of *Advanced Color Imaging on the Mac OS.* The signatures for profile tags are defined in the `CMICCProfile.h` header file.

`found` A flag that is set to `true` if the profile contains the element or `false` if it doesn't.

DESCRIPTION

You cannot use this function to test whether a profile element in the profile `CM2Header` profile header exists. Instead, you must call the `CMGetProfileHeader` function, described on page 3-64, to copy the profile header and read its contents.

CMCountProfileElements

To count the number of elements in a profile, use the `CMCountProfileElements` function.

```
pascal CMError CMCountProfileElements
                     (CMProfileRef prof,
                      unsigned long *elementCount);
```

`prof` A reference to the profile. For information on profile references, see "Profile Reference" on page 3-43.

`elementCount`
A one-based count of the number of elements. The ColorSync Manager returns this number if the function completes successfully.

DESCRIPTION

Every element in the profile outside the header is counted. A profile may contain tags that are references to other elements. These tags are included in the count. For information about profiles and their tags, see "Profile Properties" in the chapter "Introduction to the ColorSync Manager" in *Advanced Color Imaging on the Mac OS.*

CMGetProfileElement

To obtain the element data given the element's tag signature, use the `CMGetProfileElement` function.

```
pascal CMError CMGetProfileElement (CMProfileRef prof, OSType tag,
                        unsigned long *elementSize,
                        void *elementData);
```

`prof` A reference to the profile containing the target element. For information on profile references, see "Profile Reference" on page 3-43.

`tag` The tag signature for the element in question. The tag identifies the element. For a complete list of the public tag signatures a profile may contain, including a description of each tag, refer to the *International Color Consortium Profile Format Specification.* The signatures for profile tags are defined in the `CMICCProfile.h` header file.

`elementSize`
On entry, the size of the element data to be copied. Specify `NULL` to copy the entire element data. To obtain a portion of the element data, specify the number of bytes to be copied.

On return, the size of the data returned.

`elementData`

A pointer to the memory you allocated to hold the copy of the returned element data. On return, this buffer holds the element data.

To obtain the element size in the `elementSize` parameter without copying the element data to this buffer, specify `NULL` for this parameter.

DESCRIPTION

Before you call the `CMGetProfileElement` function to obtain the element data for a specific element, you must know the size in bytes of the element data in order to allocate a buffer to hold the returned data.

The `CMGetProfileElement` function serves two purposes. If you don't know the size of the element you want to obtain, you can call this function to get the element size, then call the function again to obtain the element data. Both times you call the function, you specify the reference to the profile containing the element in the `prof` parameter and the tag signature of the element in the `tag` parameter.

To obtain the element data size, call the `CMGetProfileElement` function specifying a pointer to an unsigned long data type in the `elementSize` field and a `NULL` value in the `elementData` field.

After you obtain the element size, you should allocate a buffer large enough to hold the returned element data, then call the `CMGetProfileElement` function again, specifying `NULL` in the `elementSize` parameter to copy the entire element data and a pointer to the data buffer in the `elementData` parameter.

To copy only a portion of the element data beginning from the first byte, allocate a buffer the size of the number of bytes of element data you want to obtain and specify the number of bytes to be copied in the `elementSize` parameter. In this case, on return the `elementSize` parameter contains the size in bytes of the element data actually returned.

SEE ALSO

You cannot use the `CMGetProfileElement` function to copy a portion of element data beginning from an offset into the data. To copy a portion of the element

data beginning from any offset, use the `CMGetPartialProfileElement` function on page 3-65.

You cannot use this function to obtain a portion of a profile element in the profile `CM2Header` profile header. Instead, you must call the `CMGetProfileHeader` function, described on page 3-64, to copy the profile header and read its contents.

CMGetProfileHeader

To obtain the profile header for a specific profile, use the `CMGetProfileHeader` function.

```
pascal CMError CMGetProfileHeader (CMProfileRef prof,
                          CMAppleProfileHeader *header);
```

`prof` A reference to the profile whose header is to be copied. For information on profile references, see “Profile Reference” on page 3-43.

`header` A copy of the profile header. Depending on the profile version, this may be a ColorSync 2.0 or 1.0 header. For information about the ColorSync 2.0 profile header structure, see “Profile 2.0 Header Structure for the ColorSync Manager” on page 3-26. For information about the ColorSync 1.0 header, see “Profile Header for ColorSync 1.0” on page 3-45 and the appendix, “ColorSync Manager Backward Compatibility.”

DESCRIPTION

The `CMGetProfileHeader` function returns the header for a ColorSync 2.0 or ColorSync 1.0 profile. To return the header, the function uses a union of type `CMAppleProfileHeader`, described on page 3-45, with variants for version 1.0 and 2.0 headers.

A 32-bit version value is located at the same offset in either header. For ColorSync 2.0, this is the `NumVersion` field. For ColorSync 1.0, this is the `applProfileVersion` field. You can inspect the version and interpret the remaining header fields depending on the profile version.

SEE ALSO

To copy a profile header to a profile after you modify the header's contents, use the `CMSetProfileHeader` function, described on page 3-72.

CMGetPartialProfileElement

To obtain a portion of the element data, given a reference to the profile containing the element and the element's tag signature, use the `CMGetPartialProfileElement` function.

```
pascal CMError CMGetPartialProfileElement
                     (CMProfileRef prof, OSType tag,
                      unsigned long offset,
                      unsigned long *byteCount,
                      void *elementData);
```

`prof` A reference to the profile containing the target element. For information on profile references, see "Profile Reference" on page 3-43.

`tag` The tag signature for the element in question. For a complete list of the tag signatures a profile may contain, including a description of each tag, refer to the *International Color Consortium Profile Format Specification*. The signatures for profile tags are defined in the `CMICCProfile.h` header file.

`offset` Beginning from the first byte of the element data, the offset from which to begin copying the element data.

`byteCount` On entry, the number of bytes of element data to copy beginning from the offset specified by the `offset` parameter. On return, the number of bytes actually copied.

`elementData`

The buffer to hold the element data to be copied.

DESCRIPTION

The `CMGetPartialProfileElement` function allows you to copy any portion of the element data beginning from any offset into the data. For the

CMGetPartialProfileElement function to copy the element data and return it to you, your application must allocate a buffer in memory to hold the data.

You cannot use this function to obtain a portion of a profile element in the profile's CM2Header header. Instead, you must call the CMGetProfileHeader function, described on page 3-64, to copy the profile header and read its contents.

CMGetIndProfileElementInfo

To obtain the element tag and data size of an element by index, use the CMGetIndProfileElementInfo function.

```
pascal CMError CMGetIndProfileElementInfo
                    (CMProfileRef prof, unsigned long index,
                     OSType *tag, unsigned long *elementSize,
                     Boolean *refs);
```

prof — A reference to the profile containing the element. For information on profile references, see "Profile Reference" on page 3-43.

index — A one-based element index within the range returned as the elementCount parameter of the CMCountProfileElements function.

tag — The tag signature of the element corresponding to the index. The ColorSync Manager returns the tag if the function completes successfully.

elementSize — The size in bytes of the element data corresponding to the tag. The ColorSync Manager returns the size if the function completes successfully.

refs — A flag that is set to true if more than one tag in the profile refers to element data associated with the tag corresponding to the index.

DESCRIPTION

Before calling this function, you must call the `CMCountProfileElements` function, described on page 3-61, that returns the total number of elements in the profile as the `elementCount` parameter. The number you specify for the `CMGetIndProfileElementInfo` function's `index` parameter must be in the range of 1 to `elementCount`. The index order of elements is determined internally by the ColorSync Manager and is not publicly defined.

You might want to call this function, for example, to print out the contents of a profile.

CMGetIndProfileElement

To obtain the element data corresponding to a particular index, use the `CMGetIndProfileElement` function.

```
pascal CMError CMGetIndProfileElement
                     (CMProfileRef prof,
                      unsigned long index,
                      unsigned long *elementSize,
                      void *elementData);
```

`prof` — A reference to the profile containing the element. For information on profile references, see "Profile Reference" on page 3-43.

`index` — The index of the element whose data you want to obtain. This is a one-based element index within the range returned as the `elementCount` parameter of the `CMCountProfileElements` function.

`elementSize`

On entry, the size of the element data to be copied. Specify `NULL` to copy the entire element data. To obtain a portion of the element data, specify the number of bytes to be copied.

On return, the size of the element data actually copied.

`elementData`
: A pointer to the memory you allocated to hold the copy of the returned element data. On return, this buffer holds the element data.

  To obtain the element size in the `elementSize` parameter without copying the element data to this buffer, specify `NULL` for this parameter.

DESCRIPTION

Before you call the `CMGetIndProfileElement` function to obtain the element data for an element at a specific index, you must know the size in bytes of the element data in order to allocate a buffer large enough to hold the returned data.

You can call the `CMGetIndProfileElementInfo` function, described on page 3-66, to obtain the data size of an element given the element's index. Alternatively, you can call the `CMGetIndProfileElement` function specifying a pointer to an unsigned long data type in the `elementSize` field and a `NULL` value in the `elementData` field.

After you get the size of the element data, you should allocate a buffer to hold the returned element data, then call the `CMGetIndProfileElement` function specifying `NULL` in the `elementSize` parameter to copy the entire element data and a pointer to the data buffer in the `elementData` parameter.

To copy only a portion of the element data beginning from the first byte, allocate a buffer the size of the number of bytes of element data you want to obtain and specify the number of bytes to be copied in the `elementSize` parameter. On return the `elementSize` parameter contains the size in bytes of the element data actually copied.

SEE ALSO

Before calling this function, you should call the `CMCountProfileElements` function, described on page 3-61, that returns the total number of elements in the profile as the `elementCount` parameter.

CMSetProfileElementSize

To reserve the element data size for a specific tag before setting the element data, use the `CMSetProfileElementSize` function.

```
pascal CMError CMSetProfileElementSize
                     (CMProfileRef prof,
                      OSType tag,
                      unsigned long elementSize);
```

`prof` A reference to the profile in which the element data size is to be reserved. For information on profile references, see "Profile Reference" on page 3-43.

`tag` The tag signature for the element whose size is to be reserved. The tag identifies the element. For a complete list of the tag signatures a profile may contain, including a description of each tag, refer to the *International Color Consortium Profile Format Specification*. The signatures for profile tags are defined in the `CMICCProfile.h` header file.

`elementSize`
The total size in bytes to be reserved for the element data.

DESCRIPTION

Your application may use the `CMSetProfileElementSize` function to reserve the size of element data for a specific tag before you call the `CMSetPartialProfileElement` function, described next, to set the element data. The most efficient way to set a large amount of element data when you know the size of the data is to first set the size, then call the `CMSetPartialProfileElement` function to set each of the data segments. Calling the `CMSetProfileElementSize` function first eliminates the need for the ColorSync Manager to repeatedly increase the size for the data each time you call the `CMSetPartialProfileElement` function.

In addition to reserving the element data size, the `CMSetProfileElementSize` function sets the element tag, if it does not already exist.

CMSetPartialProfileElement

To set part of the element data for a specific tag, use the `CMSetPartialProfileElement` function.

```
pascal CMError CMSetPartialProfileElement
                     (CMProfileRef prof,
                      OSType tag,
                      unsigned long offset,
                      unsigned long byteCount,
                      void *elementData);
```

`prof` A reference to the profile containing the tag for which the element data is to be set. For information on profile references, see "Profile Reference" on page 3-43.

`tag` The tag signature for the element whose data is to be set. The tag identifies the element. For a complete list of the tag signatures a profile may contain, including a description of each tag, refer to the *International Color Consortium Profile Format Specification.* The signatures for profile tags are defined in the `CMICCProfile.h` header file.

`offset` The offset of the existing element data to which to begin transferring the new element data.

`byteCount` The number of bytes of element data to transfer.

`elementData`
The buffer containing the element data to be transferred.

DESCRIPTION

You can use the `CMSetPartialProfileElement` function to set the data for an element when the amount of data is large and you need to copy it to the profile in segments.

After you set the element size, you can call `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.

If you know the size of the element data, you should call `CMSetProfileElementSize` (page 3-69) to reserve it before you call the `CMSetPartialProfileElement` function to set element data in segments. Setting the size first avoids the extensive overhead required to increase the size for the element data with each call to append another segment of data.

SEE ALSO

To copy the entire data for an element as a single operation when the amount of data is small enough to allow this, call the `CMSetProfileElement` function described on page 3-71.

CMSetProfileElement

To set or replace the element data for a specific tag, use the `CMSetProfileElement` function.

```
pascal CMError CMSetProfileElement
                     (CMProfileRef prof, OSType tag,
                      unsigned long elementSize,
                      void *elementData);
```

`prof` A reference to the profile containing the tag for which the element data is to be set. For information on profile references, see "Profile Reference" on page 3-43.

`tag` The tag signature for the element whose data is to be set. For a complete list of the tag signatures a profile may contain, including a description of each tag, refer to the *International Color Consortium Profile Format Specification*. See "Profiles" in the chapter "Introduction to the ColorSync Manager" of *Advanced Color Imaging on the Mac OS* for information on how to obtain this document. The signatures for profile tags are defined in the `CMICCProfile.h` header file.

`elementSize`
The size in bytes of the element data to be set.

`elementData`
: A pointer to the buffer containing the element data to be transferred to the profile.

DESCRIPTION

The `CMSetProfileElement` function replaces existing element data if an element with the specified tag is already present in the profile. Otherwise, it sets the element data for a new tag. Your application is responsible for allocating memory for the buffer to hold the data to be transferred.

CMSetProfileHeader

To set the header for a specific profile, use the `CMSetProfileHeader` function.

```
pascal CMError CMSetProfileHeader (CMProfileRef prof,
                          const CMAppleProfileHeader *header);
```

`prof`
: A reference to the profile whose header is to be set. For information on profile references, see "Profile Reference" on page 3-43.

`header`
: The new header for the profile.

DESCRIPTION

You can use the `CMSetProfileHeader` function to set a header for a version 1.0 or a version 2.0 ColorSync profile. Before you call this function, you must set the values for the header, depending on the version of the profile. For a version 2.0 profile, you use the `CM2Header` data structure, described "Profile 2.0 Header Structure for the ColorSync Manager" on page 3-26. For a version 1.0 profile, you use the `CMHeader` data structure, described in "Profile Header for ColorSync 1.0" on page 3-45 and discussed in the appendix, "ColorSync Manager Backward Compatibility." You pass the header you supply in the `CMAppleProfileHeader` union which is described in "Apple Profile Header" on page 3-26.

CMSetProfileElementReference

To add a tag to a profile to refer to data corresponding to a previously set element, use the `CMSetProfileElementReference` function.

```
pascal CMError CMSetProfileElementReference (CMProfileRef prof,
                        OSType elementTag, OSType referenceTag);
```

`prof` A reference to the profile to which the new tag is to be added. For information on profile references, see “Profile Reference” on page 3-43.

`elementTag` The original element’s signature tag corresponding to the element data to which the new tag will refer.

`referenceTag` The new tag signature to be added to the profile to refer to the element data corresponding to `elementTag`.

DESCRIPTION

After the `CMSetProfileElementReference` function completes successfully, the specified profile will contain more than one tag corresponding to a single piece of data. All of these tags are of equal importance. Your application may set a reference to an element that was originally a reference itself without circularity.

If you call `CMSetProfileElement` (page 3-71) subsequently for one of the tags acting as a reference to another tag’s data, then the element data you provide is set for the tag and the tag is no longer considered a reference. Instead, the tag corresponds to its own element data and not that of another tag.

CMRemoveProfileElement

To remove an element corresponding to a specific tag, use the `CMRemoveProfileElement` function.

```
pascal CMError CMRemoveProfileElement (CMProfileRef prof,
                        OSType tag);
```

`prof` A reference to the profile containing the tag to be removed. For information on profile references, see "Profile Reference" on page 3-43.

`tag` The tag signature for the element to be removed.

DESCRIPTION

The `CMRemoveProfileElement` function deletes the tag as well as the element data from the profile.

CMGetScriptProfileDescription

To obtain the name of a profile and the script code identifying the language in which the profile name is specified, use the `CMGetScriptProfileDescription` function.

```
pascal CMError CMGetScriptProfileDescription(CMProfileRef prof,
                          Str255 name, ScriptCode *code);
```

`prof` A reference to the profile whose profile name and script code you want to obtain.

`name` The profile name, returned if the function completes successfully.

`code` The script code, returned if the function completes successfully.

DESCRIPTION

The element data of the text description tag, whose signature is `'desc'`, specifies the profile name and script code. The profile name is returned as a Pascal string. This function returns that information to you so that your application does not need to obtain and parse the element data, which contains other data such as the name in UniCode format.

Matching Colors Using the High-Level Functions

The following high-level ColorSync Manager functions that you use to perform color matching acknowledge and use QuickDraw.

NCMBeginMatching

To set up a high-level ColorSync matching session that acknowledges QuickDraw operations, use the `NCMBeginMatching` function.

```
pascal CMError NCMBeginMatching (CMProfileRef src,
                                 CMProfileRef dst,
                                 CMMatchRef *myRef);
```

`src` The source profile for the matching session. To indicate the ColorSync system profile, specify a `NULL` value. For information on profile references, see “Profile Reference” on page 3-43.

`dst` The destination profile for the matching session. To indicate the ColorSync system profile, specify a `NULL` value. For information on profile references, see “Profile Reference” on page 3-43.

`myRef` A reference to the high-level matching session.

DESCRIPTION

The `NCMBeginMatching` function sets up a high-level matching session using QuickDraw, telling the ColorSync Manager to match all colors drawn to the current graphics device using the specified source and destination profiles.

The `NCMBeginMatching` function returns a reference to the color-matching session. You must later pass this reference to the `CMEndMatching` function, described next, to conclude the session.

The source and destination profiles define how the match is to occur. Passing `NULL` for either the source or destination profile is equivalent to passing the system profile. If the current device is a screen device, matching to all screen devices occurs.

The `NCMBeginMatching` and `CMEndMatching` functions can be nested. In such cases, the ColorSync Manager first matches to the most recently added profiles

first. Therefore, if you want to use the `NCMBeginMatching`-`CMEndMatching` pair to perform a page preview, which typically entails color matching from a source device (scanner) to a destination device (printer) to a preview device (display), you would first call `NCMBeginMatching` with the printer-to-display profiles, and then call `NCMBeginMatching` with the scanner-to-printer profiles. The ColorSync Manager then matches all drawing from the scanner to the printer and then back to the display. The print preview process entails multiprofile transformations, and the ColorSync Manager low-level functions which include the use of concatenated profiles well suited to print preview processing offer an easier and faster way to do this.

Note

If you call `NCMBeginMatching` before drawing to the screen's graphics device (as opposed to an offscreen device), you must call `CMEndMatching` to finish a matching session and before calling `WaitNextEvent` or any other routine (such as Window Manager routines) that could draw to the screen. Failing to do so will cause unwanted matching to occur. Furthermore, if a device has color matching enabled, you cannot call the `CopyBits` procedure to copy from it to itself unless the source and destination rectangles are the same. ◆

Even if you call the `NCMBeginMatching` function before calling the QuickDraw `DrawPicture` function, the ColorSync picture comments such as `cmEnableMatching` and `cmDisableMatching` are not acknowledged. For the ColorSync Manager to recognize these comments and allow them to be used, you must call the `NCMDrawMatchedPicture` function for color matching using picture comments.

This function causes matching for the specified devices rather than for the current color graphics port.

SEE ALSO

The high-level color-matching function `NCMBeginMatching` that uses QuickDraw performs color matching in a manner acceptable to most applications. However, if your application needs a finer level of control over color matching, it can use the low-level matching functions described in "Matching Colors Using the Low-Level Functions Without QuickDraw," beginning on page 3-80.

For background information on graphics devices, see *Inside Macintosh: Imaging With QuickDraw.*

CMEndMatching

To conclude a high-level QuickDraw matching session initiated by a previous call to the `NCMBeginMatching` function, use the `CMEndMatching` function.

```
pascal void CMEndMatching (CMMatchRef myRef);
```

`myRef` A reference to the matching session to end. This reference was previously created and returned by a call to `NCMBeginMatching` function.

DESCRIPTION

The `CMEndMatching` function releases private memory allocated for the high-level matching session to be concluded.

After you call the `NCMBeginMatching` function and before you call `CMEndMatching` to end the session, embedded color-matching picture comments, such as `cmEnableMatching` and `cmDisableMatching`, are not acknowledged. After you call `CMEndMatching`, processing reverts to its previous state.

CMEnableMatchingComment

To insert a comment in the currently open picture that turns matching on or off, use the `CMEnableMatchingComment` function.

```
pascal void CMEnableMatchingComment (Boolean enableIt);
```

`enableIt` A flag that directs the ColorSync Manager to generate a `cmEnableMatching PicComment` comment if `true` or a `cmDisbleMatching PicComment` comment if `false`.

Using Embedded Profiles With QuickDraw

NCMDrawMatchedPicture

To match a picture's colors to a destination device's color gamut as the picture is drawn, use the `NCMDrawMatchedPicture` function.

```
pascal void NCMDrawMatchedPicture (PicHandle myPicture,
                    CMProfileRef dst,
                    Rect *myRect);
```

`myPicture` The QuickDraw picture whose colors are to be matched.

`dst` The profile of the destination device. To indicate the system profile, specify a `NULL` value.

`myRect` The destination rectangle for rendering the picture specified by `myPicture`.

DESCRIPTION

The `NCMDrawMatchedPicture` function operates in the context of the current color graphics port. This function sets up and takes down a color-matching session. It automatically matches all colors in a picture to the destination profile for a destination device as the picture is drawn. It uses the ColorSync system profile as the initial source profile and any embedded profiles thereafter. (Because color-matching picture comments embedded in the picture to be matched are recognized, embedded profiles are used.)

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.

The picture will be drawn with matched colors to all screen graphics devices. If the current graphics device is not a screen device, matching will occur for that graphics device only.

If the current port is an original QuickDraw graphics port, then calling this function is equivalent to calling `DrawPicture`, in which case, no color matching occurs.

NCMUseProfileComment

To automatically embed a profile into an open picture, use the `NCMUseProfileComment` function.

```
pascal CMError NCMUseProfileComment (CMProfileRef prof,
                                     unsigned long flags);
```

`prof` A reference to the profile to be embedded. For information on profile references, see "Profile Reference" on page 3-43.

`flags` Reserved for future use.

DESCRIPTION

The `NCMUseProfileComment` function automatically generates the picture comments required to embed the specified profile into the open picture. This function performs all the tasks that your application would need to do to embed a profile. 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.

IMPORTANT

You can use this function to embed all types of profiles in an image, including device-linked profiles, but not abstract profiles. You cannot use this function to embed ColorSync 1.0 profiles in an image. ▲

The `NCMUseProfileComment` function precedes the profile it embeds with a picture comment of `kind cmBeginProfile`. 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. 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`. 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.

Matching Colors Using the Low-Level Functions Without QuickDraw

This section describes the functions you use to perform color matching using the ColorSync Manager without QuickDraw. To use the low-level functions, you first create a color-matching world, which establishes how matching will take place between the given profiles.

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 high-level functions that use 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.

Once you create a color world, it persists until you dispose of it, independent of the functions for which you use it. High-level functions that take source and destination profile reference parameters are state based, whereas the low-level functions are not.

NCWNewColorWorld

To create a color world for color matching based on the source and destination profiles, use the `NCWNewColorWorld` function.

```
pascal CMError NCWNewColorWorld (CMWorldRef *cw, CMProfileRef src,
                          CMProfileRef dst);
```

`cw` A reference to a matching session that the ColorSync Manager returns if the function completes successfully. You pass this reference to other functions that use the color world.

`src` A reference to the source profile to be used in the color-matching world. This profile's `dataColorSpace` element corresponds to the source data type for subsequent calls to functions that use this color world. For information on profile references, see "Profile Reference" on page 3-43.

`dst` A reference to the destination profile to be used in the color-matching world. This profile's `dataColorSpace` element corresponds to the destination data type for subsequent calls to functions using this color world. For information on profile references, see "Profile Reference" on page 3-43.

DESCRIPTION

You must set up a color world before your application can perform low-level color-matching or color-checking operations. To set up a color world in which these operations can occur, your application can call the `NCWNewColorWorld` function after obtaining references to the profiles to be used as the source and destination profiles for the color world. The following rules govern the types of profiles allowed:

- You may specify a device profile or a color space conversion profile for the source and destination profiles.
- You may not specify a device-linked profile or an abstract profile for either the source profile or the destination profile.
- You may specify the system profile explicitly by reference or by giving `NULL` for either the source profile or the destination profile.

You should call the `CMCloseProfile` function, described on page 3-51, for both the source and destination profiles to dispose of their references after the `NCWNewColorWorld` function completes execution.

When a color-matching or color-checking function is called using this color world, to determine the CMM to use for the session, the ColorSync Manager follows the CMM selection arbitration scheme described in the chapter "Introduction to the ColorSync Manager" in *Advanced Color Imaging on the Mac OS.*

For a brief description of a color world, see "Matching Colors Using the Low-Level Functions Without QuickDraw" on page 3-80.

SEE ALSO

The `CWConcatColorWorld` function described on page 3-82 also allocates a color world reference of type `CMWorldRef`.

CWConcatColorWorld

To set up a color world that includes a set of profiles for various color transformations among devices in a sequence, use the `CWConcatColorWorld` function.

```
pascal CMError CWConcatColorWorld (CMWorldRef *cw,
                        CMConcatProfileSet *profileSet);
```

`cw` A reference to a color world that the ColorSync Manager returns if the function completes successfully. You pass this reference to other functions that use the color world for color-matching and color-checking sessions.

`profileSet` An array of profiles describing the processing to be carried out. The array is in processing order—source through destination. See "Concatenated Profile Set Structure" on page 3-30 for the definition of the `CMConcatProfileSet` structure.

DESCRIPTION

The CWConcatColorWorld function sets up a session for color processing that includes a set of profiles. The array of profiles is in processing order—source through destination. Your application passes the function a data structure of type CMConcatProfileSet to identify the profile array.

The keyIndex field of the CMConcatProfileSet data structure specifies the index of the profile within the profile array whose preferred CMM is to be used for the entire color-matching or color-checking session. The profile header's CMMType field specifies the CMM. This CMM will fetch the profile elements necessary for the session.

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. See "Profile 2.0 Header Structure for the ColorSync Manager," beginning on page 3-26 for more information on the use of flags.

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.

The following rules govern the profiles you may specify in the profile array pointed to by the profileSet parameter for use with the CWConcatColorWorld function:

- In the profile array, you can pass in one or more profiles, but you must specify at least one profile. If you specify only one profile, it must be a device-linked profile. If you specify a device-linked profile, you cannot specify any other profiles in the profiles array; a device-linked profile must be used alone.
- In the profile array, you can specify an abstract profile anywhere in the sequence other than as the first or last profile.
- For the first and last profiles, you can specify device profiles or color space conversion profiles.
- You cannot specify NULL to indicate the system profile.
- If you specify a color space profile in the middle of the profile sequence, it is ignored by the Apple-supplied CMM.

You should call the `CMCloseProfile` function, described on page 3-51, for each profile to dispose of its reference after the `CWConcatColorWorld` function completes execution.

For a brief description of a color world, see “Matching Colors Using the Low-Level Functions Without QuickDraw” on page 3-80.

SEE ALSO

Instead of passing in an array of profiles, you can specify a device-linked profile. For information on how to create a device-linked profile, see the `CWNewLinkProfile` function, which is described next.

CWNewLinkProfile

To create a device-linked profile, use the `CWNewLinkProfile` function.

```
pascal CMError CWNewLinkProfile (CMProfileRef *prof,
                    const CMProfileLocation *targetLocation,
                    CMConcatProfileSet *profileSet);
```

`prof` A reference to the new device-linked profile. For information on profile references, see “Profile Reference” on page 3-43. The ColorSync Manager returns this reference if the function completes successfully.

`targetLocation` The location specification for the resulting profile. The ColorSync Manager returns this value if the function completes successfully. A device-linked profile cannot be a temporary profile: that is, it cannot have a location type of `cmNoProfileBase`.

`profileSet` An array of profiles describing the processing to be carried out. The array is in processing order—source through destination. For a description of the `CMConcatProfileSet` data type, see “Concatenated Profile Set Structure” on page 3-30.

DESCRIPTION

You can use this function to create a new single profile containing a set of profiles and pass the device-linked profile to the CWConcatColorWorld function, described on page 3-82, instead of specifying each profile in an array. A device-linked profile provides a means of storing in concatenated format a series of device profiles and nondevice profiles that are used repeatedly in the same sequence.

The zero-based keyIndex field of the CMConcatProfileSet data structure, described in "Concatenated Profile Set Structure" on page 3-30, specifies the index of the profile within the device-linked profile whose preferred CMM is to be used for the entire color-matching or color-checking session. The profile header's CMMType field specifies the CMM. This CMM will fetch the profile elements necessary for the session.

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 profiles that follow in the sequence are ignored. The quality flag setting is stored in the flag field of the profile header. See "Profile 2.0 Header Structure for the ColorSync Manager," beginning on page 3-26 for more information on the use of flags.

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.

The following rules govern the content and use of a device-linked profile:

- The first and last profiles you specify in the profiles array for a device-linked profile must be device profiles.
- You cannot include another device-linked profile in the series of profiles you specify in the profiles array.
- The only way to use a device-linked profile is to pass it to the CWConcatColorWorld function as the sole profile specified in the profileSet array.
- You cannot embed a device-linked profile in an image.
- You cannot specify NULL to indicate the system profile.

When your application is finished with the device-linked profile, it must close the profile with the CMCloseProfile function, described on page 3-51.

This function maintains privately all the profile information required by the color world for color-matching and color-checking sessions. Therefore, you should call the `CMCloseProfile` function for each profile used to build a device-linked profile to dispose of their references after the `CWNewLinkProfile` function completes execution.

For a brief description of a color world, see "Matching Colors Using the Low-Level Functions Without QuickDraw" on page 3-80.

CWDisposeColorWorld

To release the private storage associated with a color world when your application has finished using the color world, use the `CWDisposeColorWorld` function.

```
pascal void CWDisposeColorWorld (CMWorldRef cw);
```

`cw` A color world reference. See "Color World Reference" on page 3-44.

SEE ALSO

The `NCWNewColorWorld` function described on page 3-81 and the `CWConcatColorWorld` function described on page 3-82 both allocate color world references of type `CMWorldRef`.

The `CWMatchColors` function described on page 3-97, the `CWCheckColors` function described on page 3-98, the `CWMatchBitmap` function described on page 3-92, and the `CWCheckBitMap` function described on page 3-95 use color worlds. You must dispose of the color worlds that these functions use when your application is finished with them.

CMGetCWInfo

To obtain information about the Color Management Modules (CMMs) used for a specific color world, use the `CMGetCWInfo` function.

```
pascal CMError CMGetCWInfo (CMWorldRef cw, CMCWInfoRecord *info);
```

`cw` A reference to the color world about which you want information. See "Color World Reference" on page 3-44.

`info` A pointer to a structure of type `CMCWInfoRecord`, described on page 3-31, that your application supplies. The ColorSync Manager returns information in this structure describing the number and kind of CMMs involved in the matching session and the CMM type and version of each CMM used.

DESCRIPTION

To learn whether one or two CMMs are used for color matching and color checking in a given color world and to obtain the CMM type and version number of each one used, your application must first obtain a reference to the color world. To obtain a reference to a ColorSync color world, you must create the color world using the `NCWNewColorWorld` function or the `CWConcatColorWorld` function.

The source and destination profiles you specify when you create a color world identify their preferred CMMs, and you explicitly identify the profile whose CMM is to be used for a device-linked profile or a concatenated color world. However, you cannot be certain if the specified CMM will be used until the ColorSync Manager determines internally if the CMM is available and able to perform the requested function. For example, in some cases the Apple-supplied default CMM will be used.

The `CMGetCWInfo` function identifies the CMM or CMMs to be used. Your application must allocate a data structure of type `CMCWInfoRecord` and pass a pointer to it in the `info` parameter. The `CMGetCWInfo` function returns the color world information in this structure. The structure includes a `cmmCount` field identifying the number of CMMs to be used and an array of two members containing structures of type `CMMInfoRecord` described on page 3-32. The `CMGetCWInfo` function returns information in one or both of the CMM information records depending on whether one or two CMMs are used.

For a brief description of a color world, see “Matching Colors Using the Low-Level Functions Without QuickDraw” on page 3-80.

SEE ALSO

The `NCWNewColorWorld` function described on page 3-81 and the `CWConcatColorWorld` function described on page 3-81 both allocate color world references of type `CMWorldRef`.

CWMatchPixMap

To match a pixel map in place using a given color world, use the `CWMatchPixMap` function.

```
pascal CMError CWMatchPixMap (CMWorldRef cw, PixMap *myPixMap,
                     CMBitmapCallBackUPP progressProc,
                     void *refCon);
```

`cw` A reference to the color world in which the matching is to occur. See “Color World Reference” on page 3-44.

`myPixMap` The pixel map to be matched. A pixel map is a QuickDraw structure describing pixel data. The pixel map must be nonrelocatable; to ensure this, you should lock the handle to the pixel map before you call this function.

`progressProc`

A calling-program-supplied callback function that allows your application to monitor progress or abort the operation as the pixel map colors are matched.

The Apple-supplied CMM calls your function approximately every half-second unless color matching occurs quickly enough to warrant not calling your function; this happens when there is a small amount of data to be matched.

If the function returns a result of `true`, the operation is aborted. Specify `NULL` for this parameter if your application will not monitor the pixel map color matching. For information on the callback function and its type definition, see the description of the `MyCMBitmapCallBackProc` on page 3-134.

`refCon` A reference constant for application data that is passed as a parameter to calls to `progressProc`.

DESCRIPTION

The `CWMatchPixMap` function matches a pixel map in place using the profiles specified by the given color world. The preferred CMM, as determined by the ColorSync Manager based on the color world configuration, is called to perform the color matching.

If the preferred CMM is not available, then the ColorSync Manager calls the Apple-supplied default CMM to perform the matching. If the preferred CMM is available but it does not implement the `CMMatchPixMap` function, then the ColorSync Manager unpacks the colors in the pixel map to create a color list and calls the preferred CMM's `CMMatchColors` function, passing to this function the list of colors to be matched. Every CMM must support the `CMMatchColors` function.

For this function to complete successfully, the source and destination profiles' data color spaces (`dataColorSpace` field) must be RGB in order to match the data color space of the pixel map, which is implicitly RGB. For color spaces other than RGB, you should use the `CWMatchBitmap` function described on page 3-92.

If you specify a pointer to a calling program-supplied `MyCMBitmapCallBackProc` function in the `progressProc` parameter, the CMM performing the color matching calls your function to monitor progress of the session. Each time the CMM calls your function, it passes the function any data you specified in the `CWMatchPixMap` function's `refCon` parameter. If the ColorSync dispatcher performs the color matching, it calls your callback monitoring function once every scan line during this process.

You can use the reference constant to pass in any kind of data your callback function requires. For example, if your application uses a dialog box with a thermometer to inform the user of the color matching session's progress, you can use the reference constant to pass the dialog box's window reference to the

callback routine. For information about the callback function, see the `MyCMBitmapCallBackProc` function on page 3-134.

SEE ALSO

The `NCWNewColorWorld` function described on page 3-81 and the `CWConcatColorWorld` function described on page 3-81 both allocate color world references of type `CMWorldRef`.

Your application does not interact with the `CMMatchColors` function. However, if you want to know more about this function, see "ColorSync Manager Reference for Color Management Modules."

CWCheckPixMap

To check the colors of a pixel map using the profiles of a given color world to determine if the colors are in the gamut of the destination device, use the `CWCheckPixMap` function.

```
pascal CMError CWCheckPixMap (CMWorldRef cw, PixMap *myPixMap,
                        CMBitmapCallBackUPP progressProc,
                        void *refCon,
                        BitMap *resultBitMap);
```

`cw` A reference to the color world in which the color checking is to occur. See "Color World Reference" on page 3-44.

`myPixMap` The pixel map whose colors are to be checked. A pixel map is a QuickDraw structure describing pixel data. The pixel map must be nonrelocatable; to ensure this, you should lock the handle to the pixel map.

`progressProc`
A calling program-supplied callback function that allows your application to monitor progress or abort the operation as the pixel map colors are checked against the gamut of the destination device.

The Apple-supplied CMM calls your function approximately every half-second unless color checking occurs quickly enough to warrant not calling your function; this happens when there is a small amount of data to be checked. If the function returns a result of `true`, the operation is aborted. Specify `NULL` for this parameter if your application will not monitor the pixel map color checking. For information on the callback function and its type definition, see the MyCMBitmapCallBackProc function on page 3-134.

`refCon` A reference constant for application data passed as a parameter to calls to your `MyCMBitmapCallBackProc` function pointed to by `progressProc`.

`resultBitMap`
A QuickDraw bitmap in which pixels are set to 1 if the corresponding pixel of the pixel map indicated by `myPixMap` is out of gamut. Boundaries of the bitmap indicated by `resultBitMap` must equal the parameter of the pixel map indicated by the `myPixMap`.

DESCRIPTION

The `CWCheckPixMap` function performs a gamut test of the pixel data of the `myPixMap` pixel map according to the profiles corresponding to the specified color world to determine if the colors of the pixel map are within the gamut of the destination device as specified by the destination profile. The gamut test provides a preview of color matching using the specified color world.

The preferred CMM, as determined by the ColorSync Manager based on the profiles of the color world configuration, is called to perform the color matching.

If the preferred CMM is not available, then the ColorSync Manager calls the Apple-supplied default CMM to perform the matching. If the preferred CMM is available but it does not implement the `CMCheckPixmap` function, then the ColorSync Manager unpacks the colors in the pixel map to create a color list and calls the preferred CMM's `CMCheckColors` function passing to this function the list of colors to be matched. Every CMM must support the `CMCheckColors` function.

For this function to complete successfully, the source and destination profiles' data color spaces (`dataColorSpace` field) must be RGB in order to match the data color space of the pixel map, which is implicitly RGB.

If you specify a pointer to a calling program-supplied `MyCMBitmapCallBackProc` function in the `progressProc` parameter, the CMM performing the color checking calls your function to monitor progress of the session. Each time the CMM calls your function, it passes the function any data you specified in the `CWCheckPixMap` function's `refCon` parameter.

You can use the reference constant to pass in any kind of data your callback function requires. For example, if your application uses a dialog box with a thermometer to inform the user of the color checking session's progress, you can use the reference constant to pass the dialog box's window reference to the callback routine. For information about the callback function, see the `MyCMBitmapCallBackProc` function page 3-134.

You should ensure that the buffer pointed to by the `baseAddr` field of the bitmap passed in the `resultBitMap` parameter is zeroed out.

SEE ALSO

The `NCWNewColorWorld` function described on page 3-81 and the `CWConcatColorWorld` function described on page 3-81 both return color world references of type `CMWorldRef`.

CWMatchBitmap

To match the colors of a bitmap to the gamut of the destination device using the profiles specified by the color world, use the `CWMatchBitmap` function.

```
pascal CMError CWMatchBitmap (CMWorldRef cw,CMBitMap *bitMap,
                          CMBitmapCallBackUPP progressProc,
                          void *refCon,CMBitMap *matchedBitMap);
```

`cw` A reference to the color world in which the matching is to occur. See "Color World Reference" on page 3-44.

`bitMap` A pointer to the bitmap containing the source image data whose colors are to be matched.

`progressProc`
: A calling program-supplied callback function that allows your application to monitor progress or abort the operation as the bitmap colors are matched. The Apple-supplied CMM calls your function approximately every half second unless color matching occurs quickly enough to warrant not calling your function; this happens when there is a small amount of data to be matched. If the function returns a result of `true`, the operation is aborted. To match colors without monitoring the process, specify `NULL` for this field. For a description of the function your application supplies, see the `MyCMBitmapCallBack` function on page 3-134.

`refCon`
: A reference constant for application data passed through as a parameter to calls to the `progressProc` function.

`matchedBitMap`
: A pointer to the resulting matched bitmap containing the color-matched image. You must allocate the pixel buffer pointed to by the `image` field of the `CMBitMap` structure. If you specify `NULL` for `matchedBitMap`, then the source bitmap is matched in place.

DESCRIPTION

The `CWMatchBitmap` function matches a bitmap using the profiles specified by the given color world.

The ColorSync Manager dispatches this function to the preferred CMM. The ColorSync Manager determines the preferred CMM based on the color world configuration. If the color world you pass in was created by the `NCWNewColorWorld` function, it contains a source and destination profile, in which case the arbitration scheme described in the chapter “Introduction to the ColorSync Manager” in *Advanced Color Imaging on the Mac OS* is used to determine the preferred CMM. If the color world you pass in was created by the `CWConcatColorWorld` function, then the `keyIndex` field of the `CMConcatProfileSet` data structure identifies the preferred CMM. If the preferred CMM is not available, the Apple-supplied CMM is used to perform the color matching.

You should ensure that the buffer pointed to by the `image` field of the bitmap passed in the `bitMap` parameter is zeroed out before you call this function.

The following color spaces are currently supported for the CWMatchBitmap function:

- cmGraySpace
- cmGrayASpace
- cmRGB16Space
- cmRGB32Space
- cmARGB32Space
- cmCMYK32Space
- cmHSV32Space
- cmHLS32Space
- cmYXY32Space
- cmXYZ32Space
- cmLUV32Space
- cmLAB32Space

The ColorSync Manager does not explicitly support a CMY color space. However, for printers that have a CMY color space, you can use either of the following circumventions to make the adjustment:

- You can use a CMY profile, which the ColorSync Manager does support, with a CMYK color space. If you specify a CMYK color space in this case, the ColorSync Manager zeroes out the K channel to simulate a CMY color space.
- You can use an RGB color space and pass in the bitmap along with an RGB profile, then perform the conversion from RGB to CMY yourself.

For this function to complete successfully, the source profile's dataColorSpace field value and the space field value of the source bitmap pointed to by the bitMap parameter must specify the same data color space. Additionally, the destination profile's dataColorSpace field value and the space field value of the resulting bitmap pointed to by the matchedBitMap parameter must specify the same data color space.

Matching sessions set up with one of the multichannel color data types are not supported with this function.

SEE ALSO

The `NCWNewColorWorld` function described on page 3-81 and the `CWConcatColorWorld` function described on page 3-82 both allocate color world references of type `CMWorldRef`.

CWCheckBitMap

To test the colors of the pixel data of a bitmap, to determine if the colors map to the gamut of the destination device using a given color world, use the `CWCheckBitMap` function.

```
pascal CMError CWCheckBitMap (CMWorldRef cw, const CMBitMap *bitMap,
                     CMBitmapCallBackUPP progressProc,
                     void *refCon,
                     CMBitMap *resultBitMap);
```

`cw` A reference to the color world to be used for the color check. See “Color World Reference” on page 3-44.

`bitMap` The bitmap data whose colors are to be tested.

`progressProc`

A calling program-supplied callback function that allows your application to monitor progress or abort the operation as the bitmap’s colors are checked against the gamut of the destination device. The Apple-supplied CMM calls your function approximately every half-second unless color checking occurs quickly enough to warrant not calling your function; this happens when there is a small amount of data to be checked. If the function returns a result of `true`, the operation is aborted. Specify `NULL` for this parameter if your application will not monitor the bitmap color checking. For information on the callback function and its type definition, see the `MyCMBitmapCallBackProc` on page 3-134.

`refCon` A reference constant for application data passed as a parameter to calls to `progressProc`.

`resultBitMap`

A pointer to the resulting bitmap, indicating the outcome of the color check. The bitmap must have bounds equal to the parameter of the source bitmap pointed to by `bitMap`. You must allocate the pixel buffer pointed to by the `image` field of the `CMBitMap` structure and initialize the buffer to zeroes. Pixels are set to 1 if the corresponding pixel of the source bitmap indicated by `bitMap` is out of gamut. You must set the `space` field of the `CMBitMap` structure to `cmGamutResult1Space` color space storage format (see "Color Spaces" on page 3-15).

DESCRIPTION

When your application calls the `CWCheckBitMap` function, the ColorSync Manager dispatches the function to the preferred CMM. The ColorSync Manager determines the preferred CMM based on the color world configuration. If the color world you pass in was created by the `NCWNewColorWorld` function, the color world contains a source and destination profile, in which case the arbitration scheme described in "Introduction to the ColorSync Manager" in *Advanced Color Imaging on the Mac OS* is used to determine the preferred CMM. If the color world you pass in was created by the `CWConcatColorWorld` function, then the `keyIndex` field of the `CMConcatProfileSet` data structure identifies the preferred CMM. If the preferred CMM is not available, the Apple-supplied CMM is used to perform the color matching.

For this function to complete successfully, the source profile's `dataColorSpace` field value and the `space` field value of the source bitmap pointed to by the `bitMap` parameter must specify the same data color space.

SEE ALSO

The `NCWNewColorWorld` function described on page 3-81 and the `CWConcatColorWorld` function described on page 3-82 both allocate color world references of type `CMWorldRef`.

CWMatchColors

To match colors in a color list, use the `CWMatchColors` function.

```
pascal CMError CWMatchColors (CMWorldRef cw, CMColor *myColors,
                        unsigned long count);
```

`cw` A reference to the color world that describes how matching is to occur in the color-matching session. See "Color World Reference" on page 3-44.

`myColors` An array of type `CMColor`, described on page 3-40. On entry, this array contains the list of colors to be matched. On return, this array contains the list of matched colors specified in the color data space of the color world's destination profile.

`count` A one-based count of the number of colors in the color list of the `myColors` array.

DESCRIPTION

The `CWMatchColors` function matches colors according to the profiles corresponding to the specified color world. On entry, the color values in the `myColors` array are assumed to be specified in the data color space of the source profile. On return, the color values in the `myColors` array are transformed to the data color space of the destination profile.

All Color Management Module (CMM)s must support this function. The ColorSync Manager follows the arbitration scheme described in the chapter "Introduction to the ColorSync Manager" in *Advanced Color Imaging on the Mac OS* to determine which CMM is to be used for the color-matching session.

This function supports color matching sessions set up with one of the multichannel color data types.

SEE ALSO

The `NCWNewColorWorld` function described on page 3-81 and the `CWConcatColorWorld` function described on page 3-82 both create color worlds and return color-world references of type `CMWorldRef`.

CWCheckColors

To test a list of colors to see if they fall within a destination device's gamut, use the `CWCheckColors` function.

```
pascal CMError CWCheckColors (CMWorldRef cw,CMColor *myColors,
                          unsigned long count, long *result);
```

`cw` A reference to the color world describing how the test is to occur. See "Color World Reference" on page 3-44.

`myColors` An array of type `CMColor`, described on page 3-40, containing a list of colors. This function assumes the color values are specified in the data color space of the source profile.

`count` The number of colors in the array. This is a one-based count.

`result` A pointer to a buffer of long data types used as a bitfield with each bit representing a color in the array pointed to by `myColors`. Allocate enough memory to allow for 1 bit to represent each color in the `myColors` array. Bits in the `result` field are set to 1 if the corresponding color is out of gamut for the destination device. Ensure that the buffer you allocate is zeroed-out before you call this function.

DESCRIPTION

The color test provides a preview of color matching using the specified color world.

All CMMs must support this function. The ColorSync Manager follows the arbitration scheme described in the chapter "Introduction to the ColorSync Manager" in *Advanced Color Imaging on the Mac OS* to determine which CMM is to be used for the color-checking session.

The `result` bit array returns indication of whether the colors in the list are in or out of gamut for the destination profile. If a bit is set, its corresponding color falls out of gamut for the destination device. The leftmost bit in the field corresponds to the first color in the list.

This function supports matching sessions set up with one of the multichannel color data types.

SEE ALSO

The `NCWNewColorWorld` function described on page 3-81 and the `CWConcatColorWorld` function described on page 3-82 both allocate color-world references of type `CMWorldRef`.

Assigning and Accessing the System Profile File

The ColorSync Manager provides two functions that allow your application to identify a profile as the system profile and obtain a reference to that profile. These two functions replace the capability provided by the ColorSync 1.0 Profile Responder. The ColorSync system profile represents an abstract display device. The ColorSync Manager use it as the default profile and color space if your application does not specify a profile for the ColorSync Manager color matching and checking functions.

CMSetSystemProfile

To identify a profile as the current system profile, use the `CMSetSystemProfile` function.

```
pascal CMError CMSetSystemProfile (const FSSpec *profileFileSpec);
```

`profileFileSpec`
The file specification for the new system profile.

DESCRIPTION

By default, the Apple-supplied profile for the Apple 13-inch color monitor is configured as the system profile. By calling the `CMSetSystemProfile` function, your application may specify a new system profile.

An end user can use the ColorSync Manager control panel to identify a different profile as the system profile.

You can configure only a display device profile as the system profile.

SEE ALSO

To specify the profile file location, you use the `FSSpec` data type, described in *Inside Macintosh: Files.*

CMGetSystemProfile

To obtain a reference to the current system profile, use the `CMGetSystemProfile` function.

```
pascal CMError CMGetSystemProfile (CMProfileRef *prof);
```

`prof` A reference to the current system profile. The ColorSync Manager returns this reference if the function completes successfully.

DESCRIPTION

To give the system profile as a function parameter for any of several functions including `NCMBeginMatching`, `NCMDrawMatchedPicture`, and `NCWNewColorWorld`, you may specify the profile reference or you may specify `NULL`.

For all other functions, you must specify an explicit reference to the system profile. You may use this function to obtain a reference to the system profile.

There are other purposes for which your application might obtain a reference to the current system profile. For example, your application might need to get the profile name of the current system profile in order to display the name to the application user.

To identify the location of the physical file, call the `CMGetProfileLocation` function described on page 3-56.

SEE ALSO

When your application is finished using the current system profile, it must close the reference to the profile by calling the `CMCloseProfile` function, described on page 3-51.

Searching External Profiles

The ColorSync Manager profiles are stored in the Macintosh's ColorSync™ Profiles folder, which is located in the Preferences folder within the System Folder. The functions in this section allow your application to search for profiles within the ColorSync™ Profiles folder based on certain criteria and obtain references to the selected profiles and their file specifications.

IMPORTANT

Only profiles with a major version number of 0x02 are included in the profile search result. ▲

CMNewProfileSearch

To begin a new profile search, use the `CMNewProfileSearch` function.

```
pascal CMError CMNewProfileSearch (CMSearchRecord *searchSpec,
                          void *refCon, unsigned long *count,
                          CMProfileSearchRef *searchResult);
```

`searchSpec` The search specification. For a description of the information you may provide in a search record of type `CMSearchRecord` to define the search, see "Profile Search Record" on page 3-33.

`refCon` A reference constant for application data passed as a parameter to calls to the filter function. For a description of the filter function, see the `MyCMProfileFilterProc` function on page 3-136.

`count` A one-based count of profiles matching the search specification. The ColorSync Manager returns this number if the function completes successfully.

`searchResult`

A reference to the profile search result list. The ColorSync Manager returns this reference if the function completes successfully. For a description of the `CMProfileSearchRef` private data type, see "Profile Search Result Reference" on page 3-44.

DESCRIPTION

The `CMNewProfileSearch` function sets up and defines a new search identifying through the search record the elements that a profile must contain to qualify for inclusion in the search result list. The function searches the ColorSync™ Profiles folder for version 2.0 profiles that meet the criteria and returns a list of these profile in an internal private data structure whose reference is returned to you in the `searchResult` parameter.

You must provide a search record of type `CMSearchRecord` identifying the search criteria. You control which fields of the search record are used for any given search through a search bit mask whose value you set in the search record's `searchMask` field.

Among the information you can provide in the search record is a pointer to a calling program-supplied filter function to be used to eliminate profiles from the search based on additional criteria not defined by the search record. The search result reference is passed to the filter function after the search is performed. For a description of the filter function and its prototype, see the `MyCMProfileFilterProc` function page 3-136.

Your application cannot directly access the search result list. Instead, you pass the returned search result list reference to other search-related functions that allow you to use the result list.

SEE ALSO

To obtain a reference to a profile corresponding to a specific index in the list, use the `CMSearchGetIndProfile` function described on page 3-104. To obtain the file specification for a profile corresponding to a specific index in the list, use the `CMSearchGetIndProfileFileSpec` function described on page 3-105. To update the search result list, use the `CMUpdateProfileSearch` function described on page 3-102.

CMUpdateProfileSearch

To update an existing search result, use the `CMUpdateProfileSearch` function.

```
pascal CMError CMUpdateProfileSearch (CMProfileSearchRef search,
                          void *refCon, unsigned long *count);
```

`search` A reference to a search result list returned to your application when you called the `CMNewProfileSearch` function. For a description of the `CMProfileSearchRef` private data type, see “Profile Search Result Reference” on page 3-44.

`refCon` A reference constant for application data passed as a parameter to calls to the filter function specified by the original search specification. For a description of the filter function, see the `MyCMProfileFilterProc` function on page 3-136.

`count` A one-based count of the number of profiles matching the original search specification passed to the `CMNewProfileSearch` function if the function result is `noErr`. Otherwise undefined.

DESCRIPTION

After a profile search has been set up and performed through a call to the `CMNewProfileSearch` function, the `CMUpdateProfileSearch` function updates the existing search result. This is necessary if the contents of the ColorSync™ Profiles folder have changed since the original search result was created.

The search update uses the original search specification, including the filter function indicated by the search record. Data given in the `CMUpdateProfileSearch` function’s `refCon` parameter is passed to the filter function each time it is called.

Sharing a disk over a network makes it possible for the ColorSync™ Profiles folder contents to be modified at any time.

SEE ALSO

For a description of the function you call to begin a new search, see the `CMNewProfileSearch` function on page 3-101. This is the function that specifies the filter function referred to in the description of the `refCon` parameter.

CMDisposeProfileSearch

To free the private memory allocated for a profile search after your application has completed the search, use the `CMDisposeProfileSearch` function.

```
pascal void CMDisposeProfileSearch (CMProfileSearchRef search);
```

`search` A reference to the profile search result list whose private memory is to be released. For a description of the `CMProfileSearchRef` private data type, For a description of the `CMProfileSearchRef` private data type, see "Profile Search Result Reference" on page 3-44.

SEE ALSO

To set up a search, use the `CMNewProfileSearch` function described on page 3-101. To obtain a reference to a profile corresponding to a specific index in the list, use the `CMSearchGetIndProfile` function described on page 3-104. To obtain the file specification for a profile corresponding to a specific index in the list, use the `CMSearchGetIndProfileFileSpec` function described on page 3-105. To update the search result list, use the `CMUpdateProfileSearch` function described on page 3-102.

CMSearchGetIndProfile

To open the profile corresponding to a specific index into a specific search result list and obtain a reference to that profile, use the `CMSearchGetIndProfile` function.

```
pascal CMError CMSearchGetIndProfile (CMProfileSearchRef search,
                          unsigned long index,
                          CMProfileRef *prof);
```

`search` A reference to the profile search result list containing the profile whose reference you want to obtain. For a description of the `CMProfileSearchRef` private data type, see "Profile Search Result Reference" on page 3-44.

`index` The position of the profile in the search result list. This value is specified as a one-based index into the set of profiles of the search result. The number must be within the range returned as the `count` parameter of the `CMNewProfileSearch` function or the `CMUpdateProfileSearch` function if the search result was updated.

`prof` A reference to the profile associated with the specified index. The ColorSync Manager returns this reference if the function completes successfully.

SEE ALSO

Before your application can call the `CMSearchGetIndProfile` function, it must call the `CMNewProfileSearch` function to perform a profile search and produce a search result list. The search result list is a private data structure maintained by the ColorSync Manager. After your application has finished using the profile reference, it must close the reference by calling the `CMCloseProfile` function.

CMSearchGetIndProfileFileSpec

To obtain the file specification for the profile at a specific index into a search result, use the `CMSearchGetIndProfileFileSpec` function.

```
pascal CMError CMSearchGetIndProfileFileSpec
                    (CMProfileSearchRef search,
                     unsigned long index,
                     FSSpec *profileFile);
```

`search` A reference to the profile search result containing the profile whose file specification you want to obtain. For a description of the `CMProfileSearchRef` private data type, see “Profile Search Result Reference” on page 3-44.

`index` The index of the profile whose file specification you want to obtain. This is a one-based index into a set of profiles in the search result list. This number must be within the range

returned as the `count` parameter of the `CMNewProfileSearch` function or the `CMUpdateProfileSearch` function if the search result was updated.

`profileFile`

The file specification for the profile. The ColorSync Manager returns this value if the function completes successfully. For a description of the `FSSpec` data type, see *Inside Macintosh: Files.*

DESCRIPTION

Before your application can call the `CMSearchGetIndProfile` function, it must call the `CMNewProfileSearch` function to perform a profile search and produce a search result list. The search result list is a private data structure maintained by ColorSync.

The `CMSearchGetIndProfileFileSpec` function obtains the Macintosh file system file specification for a profile at a specific index in the search result list.

Converting Between Color Spaces

The ColorSync Manager includes a color conversion component that supports functions your application can call to convert a list of colors within the same base family. Color conversion, which does not require the use of color profiles, is a much simpler process than color matching.

The color conversion functions support conversion only between color spaces in the same base family, that is between the base color space and any of its derived color spaces or between two derivatives of the same base family.

You can convert a list of colors between XYZ and any of its derived color spaces, which include L*a*b*, L*u*v*, and Yxy, or between any two of the derived color spaces. You can also convert colors defined in the XYZ color space between `CMXYZColor` data types in which the color components are expressed as 16-bit unsigned values and `CMFixedXYZColor` data types in which the colors are expressed as 32-bit signed values.

You can convert a list of colors between RGB, which is the base-additive device-dependent color space, and any of its derived color spaces, such as HLS, HSV, and Gray, or between any two of the derived color spaces.

Here are brief descriptions of the XYZ color space and its derivative color spaces:

- The XYZ space, referred to as the interchange color space, is the fundamental, or base CIE-based independent color space.
- The L*a*b* color space is a CIE-based independent color space used for representing subtractive systems, where light is absorbed by colorants such as inks and dyes. The L*a*b* color space is derived from the XYZ color space. The default white point for the L*a*b* interchange space is the D50 white point.
- The L*u*v* color space is a CIE-based color space used for representing additive color systems, including color lights and emissive phosphor displays. The L*u*v* color space is derived from the XYZ color space.
- The Yxy color space expresses the XYZ values in terms of x and y chromaticity coordinates, somewhat analogous to the hue and saturation coordinates of HSV space. This allows color variation in Yxy space to be plotted on a two-dimensional diagram.
- The XYZ color space includes two XYZ data type formats. The `CMFixedXYZColor` data type uses the Fixed data type for each of the three components. Fixed is a signed 32-bit value. The `CMFixedXYZColor` data type, which is also used in the ColorSync Manager profile.The `CMXYZColor` data type uses 16-bit values for each component.

Here are brief descriptions of the RGB color space and its derivative color spaces:

- The RGB color space is a three-dimensional color space whose components are the red, green, and blue intensities that make up a given color.
- The HLS and HSV color spaces belong to the family of RGB-based color spaces, which are directly supported by most color displays and scanners.
- Gray spaces typically have a single component, ranging from black to white. The Gray color space is used for black-and-white and grayscale display and printing.

To convert colors from one color space to another, you don’t need to specify source and destination profiles. Instead, you must identify the ColorSync Manager color conversion component that is to perform the color conversion by specifying the component instance your application uses. The component instance identifies your application’s unique access path to the ColorSync

Manager color conversion component. Your application must use the Component Manager to open a connection to the color conversion component.

The color conversion component is a stand-alone component that all applications can use, including third-party CMMs. The color conversion component, which is part of the ColorSync extension, is registered at startup time but loaded only when needed.

Note
The color conversion functions do not support conversion of HiFi colors. ◆

CMXYZToLab

To convert colors specified in the XYZ color space to the L*a*b* color space, use the `CMXYZToLab` function. Both color spaces are device independent.

```
pascal ComponentResult CMXYZToLab (ComponentInstance ci,
                     const CMColor *src, const CMXYZColor *white,
                     CMColor *dst, unsigned long count);
```

`ci` The component instance identifying the connection your application owns to the color conversion component that performs the conversion. The ColorSync Manager color conversion component has a component type of `'ccnv'`, a component subtype of `'dflt'`, and a component manufacturer of `'appl'`.

`src` A pointer to an array containing the list of XYZ colors to be converted to L*a*b* colors.

`white` The reference white point.

`dst` A pointer to an array containing the list of L*a*b* colors resulting from the conversion.

`count` The number of colors to be converted.

DESCRIPTION

The `CMXYZToLab` function converts one or more colors defined in the XYZ color space to equivalent colors defined in the L*a*b* color space.

The `ci` parameter specifies the component instance identifying the path your application uses to gain access to the conversion component. A component instance is returned to your application when you call the Component Manager's `OpenComponent` or `OpenDefaultComponent` function to open a connection to the component. When you no longer need access to the conversion component, your application must call the Component Manager's `CloseComponent` function to close the connection and dispose of the private storage used by the component instance.

If your application does not require that you preserve the source color list, you can pass the pointer to the same color list array as the `src` and `dst` parameters and allow the `CMXYZToLab` function to overwrite the source colors with the resulting converted color specifications.

SEE ALSO

For information on how to open a connection to the conversion component to obtain a component instance and close the connection, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox.*

CMLabToXYZ

To convert colors specified in the L*a*b* color space to the XYZ color space, use the `CMLabToXYZ` function. Both color spaces are device independent.

```
pascal ComponentResult CMLabToXYZ (ComponentInstance ci,
                    const CMColor *src,
                    const CMXYZColor *white,
                    CMColor *dst, unsigned long count);
```

`ci` The component instance identifying the connection your application owns to the color conversion component that performs the conversion. The ColorSync Manager color

	conversion component has a component type of `'ccnv'`, a component subtype of `'dflt'`, and a component manufacturer of `'appl'`.
`src`	A pointer to a buffer containing the list of L*a*b* colors to be converted to XYZ colors.
`white`	The reference white point.
`dst`	A pointer to a buffer containing the list of colors as specified in the XYZ color space resulting from the conversion.
`count`	The number of colors to be converted.

DESCRIPTION

The `CMLabToXYZ` function converts one or more colors defined in the L*a*b color space to equivalent colors defined in the XYZ color space.

The `ci` parameter specifies the component instance identifying the path your application uses to gain access to the conversion component. A component instance is returned to your application when you call the Component Manager's `OpenComponent` or `OpenDefaultComponent` function to open a connection to the component. When you no longer need access to the conversion component, your application must call the Component Manager's `CloseComponent` function to close the connection dispose of the private storage used by the component instance.

SEE ALSO

For information on how to open a connection to the conversion component to obtain a component instance and close the connection, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox.*

CMXYZToLuv

To convert colors specified in the XYZ color space to the L*u*v* color space, use the `CMXYZToLuv` function. Both color spaces are device independent.

```
pascal ComponentResult CMXYZToLuv (ComponentInstance ci,
                          const CMColor *src,
                          const CMXYZColor *white,
                          CMColor *dst,
                          unsigned long count);
```

`ci` The component instance identifying the connection your application owns to the color conversion component that performs the conversion. The ColorSync Manager color conversion component has a component type of `'ccnv'`, a component subtype of `'dflt'`, and a component manufacturer of `'appl'`.

`src` A pointer to an array containing the list of XYZ colors to be converted to L*u*v* colors.

`white` The reference white point.

`dst` A pointer to an array containing the list of colors represented in L*u*v* color space resulting from the conversion.

`count` The number of colors to be converted.

DESCRIPTION

The `CMXYZToLuv` function converts one or more colors defined in the XYZ color space to equivalent colors defined in the L*u*v* color space.

The `ci` parameter specifies the component instance identifying the path your application uses to gain access to the conversion component. A component instance is returned to your application when you call the Component Manager's `OpenComponent` or `OpenDefaultComponent` function to open a connection to the component. When you no longer need access to the conversion component, your application must call the Component Manager's `CloseComponent` function to close the connection and dispose of the private storage used by the component instance.

If your application does not require that you preserve the source color list, you can pass the pointer to the same color list array as the `src` and `dst` parameters and allow the `CMXYZToLuv` function to overwrite the source colors with the resulting converted color specifications.

SEE ALSO

For information on how to open a connection to the conversion component to obtain a component instance and close the connection, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox.*

CMLuvToXYZ

To convert colors specified in the L*u*v* color space to the XYZ color space, use the `CMLuvToXYZ` function. Both color spaces are device independent.

```
pascal ComponentResult CMLuvToXYZ (ComponentInstance ci,
                        const CMColor *src,
                        const CMXYZColor *white,
                        CMColor *dst,
                        unsigned long count);
```

`ci` The component instance identifying the connection your application owns to the color conversion component that performs the conversion. The ColorSync Manager color conversion component has a component type of `'ccnv'`, a component subtype of `'dflt'`, and a component manufacturer of `'appl'`.

`src` A pointer to an array containing the list of L*u*v* colors to be converted.

`white` The reference white point.

`dst` A pointer to an array containing the list of colors, resulting from the conversion, as specified in the XYZ color space.

`count` The number of colors to be converted.

DESCRIPTION

The `CMLuvToXYZ` function converts one or more colors defined in the L*a*b color space to equivalent colors defined in the XYZ color space.

The `ci` parameter specifies the component instance identifying the path your application uses to gain access to the conversion component. A component instance is returned to your application when you call the Component Manager's `OpenComponent` or `OpenDefaultComponent` function to open a connection to the component. When you no longer need access to the conversion component, your application must call the Component Manager's `CloseComponent` function to close the connection dispose of the private storage used by the component instance.

SEE ALSO

For information on how to open a connection to the conversion component to obtain a component instance and close the connection, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox.*

CMXYZToYxy

To convert colors specified in the XYZ color space to the Yxy color space, use the `CMXYZToYxy` function. Both color spaces are device independent.

```
pascal ComponentResult CMXYZToYxy (ComponentInstance ci,
                           const CMColor *src,
                           CMColor *dst,
                           unsigned long count);
```

`ci` The component instance identifying the connection your application owns to the color conversion component that performs the conversion. The ColorSync Manager color conversion component has a component type of `'ccnv'`, a component subtype of `'dflt'`, and a component manufacturer of `'appl'`.

`src` A pointer to an array containing the list of XYZ colors to be converted to Yxy colors.

`dst` A pointer to an array containing the list of colors resulting from the conversion represented in the Yxy color space.

`count` The number of colors to be converted.

DESCRIPTION

The `CMXYZToYxy` function converts one or more colors defined in the XYZ color space to equivalent colors defined in the Yxy color space.

The `ci` parameter specifies the component instance identifying the path your application uses to gain access to the conversion component. A component instance is returned to your application when you call the Component Manager's `OpenComponent` or `OpenDefaultComponent` function to open a connection to the component. When you no longer need access to the conversion component, your application must call the Component Manager's `CloseComponent` function to close the connection and dispose of the private storage used by the component instance.

If your application does not require that you preserve the source color list, you can pass the pointer to the same color list array as the `src` and `dst` parameters and allow the `CMXYZToYxy` function to overwrite the source colors with the resulting converted color specifications.

SEE ALSO

For information on how to open a connection to the conversion component to obtain a component instance and close the connection, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox.*

CMYxyToXYZ

To convert colors specified in the Yxy color space to the XYZ color space, use the `CMYxyToXYZ` function. Both color spaces are device independent.

```
pascal ComponentResult CMYxyToXYZ (ComponentInstance ci,
                     const CMColor *src,
                     CMColor *dst,
                     unsigned long count);
```

`ci` The component instance identifying the connection your application owns to the color conversion component that performs the conversion. The ColorSync color conversion component has a component type of `'ccnv'`, a component subtype of `'dflt'`, and a component manufacturer of `'appl'`.

`src` A pointer to an array containing the list of Yxy colors to be converted.

`dst` A pointer to an array containing the list of colors, resulting from the conversion, as specified in the XYZ color space.

`count` The number of colors to be converted.

DESCRIPTION

The `CMYxyToXYZ` function converts one or more colors defined in the Yxy color space to equivalent colors defined in the XYZ color space.

The `ci` parameter specifies the component instance identifying the path your application uses to gain access to the conversion component. A component instance is returned to your application when you call the Component Manager's `OpenComponent` or `OpenDefaultComponent` function to open a connection to the component. When you no longer need access to the conversion component, your application must call the Component Manager's `CloseComponent` function to close the connection and dispose of the private storage used by the component instance.

If your application does not require that you preserve the source color list, you can pass the pointer to the same color list array as the `src` and `dst` parameters and allow the `CMYxyToXYZ` function to overwrite the source colors with the resulting converted color specifications.

SEE ALSO

For information on how to open a connection to the conversion component to obtain a component instance and close the connection, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox.*

CMXYZToFixedXYZ

To convert colors specified in the XYZ color space whose components are expressed as XYZ 16-bit unsigned values of type `CMXYZColor` to equivalent colors expressed as 32-bit signed values of type `CMFixedXYZColor`, use the `CMXYZToFixedXYZ` function. The XYZ color space is device independent.

```
pascal ComponentResult CMXYZToFixedXYZ (ComponentInstance ci,
                        const CMXYZColor *src,
                        CMFixedXYZColor *dst,
                        unsigned long count);
```

`ci` The component instance identifying the connection your application owns to the color conversion component for the conversion. The ColorSync Manager color conversion component has a component type of `'ccnv'`, a component subtype of `'dflt'`, and a component manufacturer of `'appl'`.

`src` A pointer to an array containing the list of XYZ colors to be converted to Fixed XYZ colors.

`dst` A pointer to an array containing the list of colors resulting from the conversion in which the colors are specified as Fixed XYZ colors.

`count` The number of colors to be converted.

DESCRIPTION

The `CMXYZToFixedXYZ` function converts one or more colors whose components are defined as XYZ colors to equivalent colors whose components are defined as Fixed XYZ colors. Fixed XYZ colors allow for 32-bit precision.

The `ci` parameter specifies the component instance identifying the path your application uses to gain access to the conversion component. A component instance is returned to your application when you call the Component Manager's `OpenComponent` or `OpenDefaultComponent` function to open a connection to the component. When you no longer need access to the conversion component, your application must call the Component Manager's `CloseComponent` function to close the connection and dispose of the private storage used by the component instance.

If your application does not require that you preserve the source color list, you can pass the pointer to the same color list array as the `src` and `dst` parameters and allow the `CMXYZToFixedXYZ` function to overwrite the source colors with the resulting converted color specifications.

SEE ALSO

For information on how to open a connection to the conversion component to obtain a component instance and close the connection, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox.*

CMFixedXYZToXYZ

To convert colors specified in XYZ color space whose components are expressed as Fixed XYZ 32-bit signed values of type `CMFixedXYZColor` to equivalent colors expressed as XYZ 16-bit unsigned values of type `CMXYZColor`, use the `CMFixedXYZToXYZ` function. The XYZ color space is device independent.

```
pascal ComponentResult CMFixedXYZToXYZ (ComponentInstance ci,
                        const CMFixedXYZColor *src,
                        CMXYZColor *dst, unsigned long count);
```

`ci` The component instance identifying the connection your application owns to the color conversion component that performs the conversion. The ColorSync Manager color conversion component has a component type of `'ccnv'`, a component subtype of `'dflt'`, and a component manufacturer of `'appl'`.

`src` A pointer to an array containing the list of Fixed XYZ colors to be converted to XYZ colors.

`dst` A pointer to an array containing the list of colors resulting from the conversion specified as XYZ colors.

`count` The number of colors to be converted.

DESCRIPTION

The `CMFixedXYZToXYZ` function converts one or more colors defined in the Fixed XYZ color space to equivalent colors defined in the XYZ color space.

The `ci` parameter specifies the component instance identifying the path your application uses to gain access to the conversion component. A component instance is returned to your application when you call the Component Manager's `OpenComponent` or `OpenDefaultComponent` function to open a connection to the component. When you no longer need access to the conversion component, your application must call the Component Manager's `CloseComponent` function to close the connection and dispose of the private storage used by the component instance.

If your application does not require that you preserve the source color list, you can pass the pointer to the same color list array as the `src` and `dst` parameters and allow the `CMFixedXYZToXYZ` function to overwrite the source colors with the resulting converted color specifications.

SEE ALSO

For information on how to open a connection to the conversion component to obtain a component instance and close the connection, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox.*

CMRGBToHLS

To convert colors specified in the RGB color space to equivalent colors defined in the HLS color space, use the `CMRGBToHLS` function. Both color spaces are device dependent.

```
pascal ComponentResult CMRGBToHLS (ComponentInstance ci,
                       const CMColor *src,
                       CMColor *dst,
                       unsigned long count);
```

`ci` The component instance identifying the connection your application owns to the color conversion component that performs the conversion. The ColorSync Manager color

conversion component has a component type of `'ccnv'`, a component subtype of `'dflt'`, and a component manufacturer of `'appl'`.

`src` A pointer to an array containing the list of RGB colors to be converted to HLS colors.

`dst` A pointer to an array containing the list of colors, resulting from the conversion, as specified in the HLS color space.

`count` The number of colors to be converted.

DESCRIPTION

The `CMRGBToHLS` function converts one or more colors defined in the RGB color space to equivalent colors defined in the HLS color space.

The `ci` parameter specifies the component instance identifying the path your application uses to gain access to the conversion component. A component instance is returned to your application when you call the Component Manager's `OpenComponent` or `OpenDefaultComponent` function to open a connection to the component. When you no longer need access to the conversion component, your application must call the Component Manager's `CloseComponent` function to close the connection and dispose of the private storage used by the component instance.

If your application does not require that you preserve the source color list, you can pass the pointer to the same color list array as the `src` and `dst` parameters and allow the `CMRGBToHLS` function to overwrite the source colors with the resulting converted color specifications.

SEE ALSO

For information on how to open a connection to the conversion component to obtain a component instance and close the connection, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox.*

CMHLSToRGB

To convert colors specified in the HLS color space to equivalent colors defined in the RGB color space, use the `CMHLSToRGB` function. Both color spaces are device dependent.

```
pascal ComponentResult CMHLSToRGB (ComponentInstance ci,
                        const CMColor *src,
                        CMColor *dst,
                        unsigned long count);
```

`ci` The component instance identifying the connection your application owns to the color conversion component that performs the conversion. The ColorSync Manager color conversion component has a component type of `'ccnv'`, a component subtype of `'dflt'`, and a component manufacturer of `'appl'`.

`src` A pointer to an array containing the list of HLS colors to be converted to RGB colors.

`dst` A pointer to an array containing the list of colors, resulting from the conversion, as specified in the RGB color space.

`count` The number of colors to be converted.

DESCRIPTION

The `CMHLSToRGB` function converts one or more colors defined in the HLS color space to equivalent colors defined in the RGB color space.

The `ci` parameter specifies the component instance identifying the path your application uses to gain access to the conversion component. A component instance is returned to your application when you call the Component Manager's `OpenComponent` or `OpenDefaultComponent` function to open a connection to the component. When you no longer need access to the conversion component, your application must call the Component Manager's `CloseComponent` function to close the connection and dispose of the private storage used by the component instance.

If your application does not require that you preserve the source color list, you can pass the pointer to the same color list array as the `src` and `dst` parameters

and allow the CMHLSToRGB function to overwrite the source colors with the resulting converted color specifications.

SEE ALSO

For information on how to open a connection to the conversion component to obtain a component instance and close the connection, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox.*

CMRGBToHSV

To convert colors specified in the RGB color space to equivalent colors defined in the HSV color space when the device types are the same, use the CMRGBToHSV function. Both color spaces are device dependent.

```
pascal ComponentResult CMRGBToHSV (ComponentInstance ci,
                       const CMColor *src,
                       CMColor *dst,
                       unsigned long count);
```

`ci` The component instance identifying the connection your application owns to the color conversion component that performs the conversion. The ColorSync Manager color conversion component has a component type of 'ccnv', a component subtype of 'dflt', and a component manufacturer of 'appl'.

`src` A pointer to an array containing the list of RGB colors to be converted to HSV colors.

`dst` A pointer to an array containing the list of colors, resulting from the conversion, as specified in the HSV color space.

`count` The number of colors to be converted.

DESCRIPTION

The CMRGBToHSV function converts one or more colors defined in the RGB color space to equivalent colors defined in the HSV color space.

The ci parameter specifies the component instance identifying the path your application uses to gain access to the conversion component. A component instance is returned to your application when you call the Component Manager's OpenComponent or OpenDefaultComponent function to open a connection to the component. When you no longer need access to the conversion component, your application must call the Component Manager's CloseComponent function to close the connection and dispose of the private storage used by the component instance.

If your application does not require that you preserve the source color list, you can pass the pointer to the same color list array as the src and dst parameters and allow the CMRGBToHSV function to overwrite the source colors with the resulting converted color specifications.

SEE ALSO

For information on how to open a connection to the conversion component to obtain a component instance and close the connection, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox.*

CMHSVToRGB

To convert colors specified in the HSV color space to equivalent colors defined in the RGB color space, use the CMHSVToRGB function. Both color spaces are device dependent.

```
pascal ComponentResult CMHSVToRGB (ComponentInstance ci,
                       const CMColor *src,
                       CMColor *dst,
                       unsigned long count);
```

ci — The component instance identifying the connection your application owns to the color conversion component that performs the conversion. The ColorSync Manager color conversion component has a component type of 'ccnv', a component subtype of 'dflt', and a component manufacturer of 'appl'.

`src`	A pointer to an array containing the list of HSV colors to be converted to RGB colors.
`dst`	A pointer to an array containing the list of colors, resulting from the conversion, as specified in the RGB color space.
`count`	The number of colors to be converted.

DESCRIPTION

The `CMHSVToRGB` function converts one or more colors defined in the HSV color space to equivalent colors defined in the RGB color space.

The `ci` parameter specifies the component instance identifying the path your application uses to gain access to the conversion component. A component instance is returned to your application when you call the Component Manager's `OpenComponent` or `OpenDefaultComponent` function to open a connection to the component. When you no longer need access to the conversion component, your application must call the Component Manager's `CloseComponent` function to close the connection and dispose of the private storage used by the component instance.

If your application does not require that you preserve the source color list, you can pass the pointer to the same color list array as the `src` and `dst` parameters and allow the `CMHSVToRGB` function to overwrite the source colors with the resulting converted color specifications.

SEE ALSO

For information on how to open a connection to the conversion component to obtain a component instance and close the connection, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox.*

CMRGBToGray

To convert colors specified in the RGB color space to equivalent colors defined in the Gray color space, use the `CMRGBToGray` function. Both color spaces are device dependent.

```
pascal ComponentResult CMRGBToGray (ComponentInstance ci,
                    const CMColor *src,
                    CMColor *dst,
                    unsigned long count);
```

`ci` The component instance identifying the connection your application owns to the color conversion component that performs the conversion. The ColorSync Manager color conversion component has a component type of `'ccnv'`, a component subtype of `'dflt'`, and a component manufacturer of `'appl'`.

`src` A pointer to an array containing the list of colors specified in RGB space to be converted to colors specified in Gray space.

`dst` A pointer to an array containing the list of colors, resulting from the conversion, as specified in the Gray color space.

`count` The number of colors to be converted.

DESCRIPTION

The `CMRGBToGray` function converts one or more colors defined in the RGB color space to equivalent colors defined in the Gray color space.

The `ci` parameter specifies the component instance identifying the path your application uses to gain access to the conversion component. A component instance is returned to your application when you call the Component Manager's `OpenComponent` or `OpenDefaultComponent` function to open a connection to the component. When you no longer need access to the conversion component, your application must call the Component Manager's `CloseComponent` function to close the connection and dispose of the private storage used by the component instance.

If your application does not require that you preserve the source color list, you can pass the pointer to the same color list array as the `src` and `dst` parameters

and allow the `CMRGBToGray` function to overwrite the source colors with the resulting converted color specifications.

SEE ALSO

For information on how to open a connection to the conversion component to obtain a component instance and close the connection, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox.*

PostScript Color-Matching Support Functions

The ColorSync Manager provides three functions that support color matching by PostScript Level 2 devices. The default Apple-supplied CMM implements these functions if the preferred CMM corresponding to the profile does not.

CMGetPS2ColorSpace

To receive the color space element data in text format usable as the parameter to the PostScript `setColorSpace` operator, which characterizes the color space of subsequent graphics data, use the `CMGetPS2ColorSpace` function.

```
pascal CMError CMGetPS2ColorSpace (CMProfileRef srcProf,
                        unsigned long flags,
                        CMFlattenUPP proc, void *refCon,
                        Boolean *preferredCMMnotfound);
```

`srcProf` A profile reference to the source profile that defines the data color space and identifies the preferred CMM. You cannot specify `NULL` to indicate the system profile. If you use the system profile, you must give an explicit reference.

`flags` Reserved for future use.

`proc` A calling program-supplied flatten function used to receive the PostScript data from the CMM. For information, see the `MyColorSyncDataTransfer` function on page 3-131.

`refCon` A reference constant for application data passed as a parameter to calls to the `MyColorSyncDataTransfer` function.

`preferredCMMnotfound`
A flag that returns `true` if the CMM corresponding to profile was not available or if it was unable to perform the function and the Apple-supplied default CMM was used. Otherwise, it returns `false`.

DESCRIPTION

The`CMGetPS2ColorSpace` function obtains the data color space element data assigned to the PostScript Level 2 color space array (`ps2CSATag)` tag in the source profile. If the tag does not exist in the profile, the CMM will create the color space array from the profile.

The CMM obtains the PostScript data from the profile and calls your low-level data transfer procedure passing the PostScript data to it. The CMM converts the data into a PostScript stream and calls your procedure as many times as necessary to transfer the data to it.

Typically, the low-level data transfer function returns this data to the calling application or device driver to be passed to a PostScript printer as an operand to the PostScript `setcolorspace` operator, which defines the color space of graphics data to follow.

The `CMGetPS2ColorSpace` function is dispatched to the CMM component specified by the source profile. If the designated CMM is not available or the CMM does not implement this function, then the ColorSync Manager dispatches the function to the Apple-supplied default CMM.

CMGetPS2ColorRenderingIntent

To receive the rendering intent element data in text format usable as the parameter to the PostScript `findRenderingIntent` operator, which specifies the color-matching option for subsequent graphics data, use the `CMGetPS2ColorRenderingIntent` function.

```
pascal CMError CMGetPS2ColorRenderingIntent
                     (CMProfileRef srcProf,  unsigned long flags,
                      CMFlattenUPP proc, void *refCon,
                      Boolean *preferredCMMnotfound);
```

`srcProf` A profile reference to the source profile. You cannot specify `NULL` to indicate the system profile. If you use the system profile, you must specify an explicit reference.

`flags` Reserved for future use.

`proc` A calling-program-supplied low-level data transfer function used to receive the PostScript data from the CMM. For information, see the `MyColorSyncDataTransfer` function on page 3-131.

`refCon` A reference constant for application data passed as a parameter to calls to the `MyColorSyncDataTransfer` function.

`preferredCMMnotfound`
A flag that returns `true` if the CMM corresponding to profile was not available or if it was unable to perform the function and the default CMM was used. Otherwise returns `false`.

DESCRIPTION

The `CMGetPS2ColorRenderingIntent` function obtains the rendering intent element data for the PostScript Level 2 rendering intent (`ps2RenderingIntentTag`) tag in the source profile. If the tag does not exist in the profile, the ColorSync Manager will create it based on the profile contents.

The CMM obtains the PostScript data from the profile and calls your low-level data transfer procedure passing the PostScript data to it. The CMM converts the data into a PostScript stream and calls your procedure as many times as necessary to transfer the data to it.

Typically, the low-level data transfer function returns this data to the calling application or device driver to be passed to a PostScript printer.

The `CMGetPS2ColorRenderingIntent` function is dispatched to the CMM component specified by the source profile. If the designated CMM is not available or the CMM does not implement this function, then ColorSync dispatches the `CMGetPS2ColorRenderingIntent` function to the default Apple-supplied CMM.

CMGetPS2ColorRendering

To receive the color rendering dictionary (CRD) element data in text usable as the parameter to the PostScript `setColorRendering` operator, which specifies the PostScript color rendering dictionary to be used for the following graphics data, use the `CMGetPS2ColorRendering` function.

```
pascal CMError CMGetPS2ColorRendering
                     (CMProfileRef srcProf,
                      CMProfileRef dstProf,
                      unsigned long flags,
                      CMFlattenUPP proc, void *refCon,
                      Boolean *preferredCMMnotfound);
```

`srcProf` A profile reference to the source profile.

`dstProf` A profile reference to the destination profile.

`flags` Reserved for future use.

`proc` A calling-program-supplied flatten function used to perform the data transfer. For information, see the `MyColorSyncDataTransfer` function on page 3-131.

`refCon` A reference constant for application data passed as a parameter to calls to the `MyColorSyncDataTransfer` function.

`preferredCMMnotfound`
A flag that returns `true` if the CMM corresponding to profile was not available or if it was unable to perform the function and the default CMM was used. Otherwise, it returns `false`.

DESCRIPTION

The `CMGetPS2ColorRendering` function obtains the color rendering dictionary (CRD) element data for the PostScript Level 2 rendering intent specified by the source profile. If the tag does not exist in the profile, the ColorSync Manager will create it based on the profile contents.

The `CMGetPS2ColorRendering` function is dispatched to the CMM component specified by the destination profile. If the designated CMM is not available or the CMM does not implement this function, then the ColorSync Manager

dispatches the `CMGetPS2ColorRendering` function to the default Apple-supplied CMM.

A profile's `ps2CRD0Tag` element data contains the CRD for perceptual rendering. A profile's `ps2CRD1Tag` contains the CRD for relative colorimetric rendering. A profile's `ps2CRD2Tag` contains the CRD for saturation rendering. A profile's `ps2CRD3Tag` contains the CRD for absolute colorimetric rendering. If the profile does not contain a CRD tag, the Apple-supplied CMM will create the CRD from the profile.

The CMM obtains the PostScript data from the profile and calls your low-level data transfer procedure passing the PostScript data to it. The CMM converts the data into a PostScript stream and calls your procedure as many times as necessary to transfer the data to it.

Typically, the low-level data transfer function returns this data to the calling application or device driver to be passed to a PostScript printer.

SEE ALSO

Before your application or device driver sends the CRD to the printer, it can call the `CMGetPS2ColorRenderingVMSize` function, described on page 3-129, to determine the virtual memory size of the CRD.

CMGetPS2ColorRenderingVMSize

To determine the virtual memory size of the color rendering dictionary for a printer profile before your application or driver obtains the CRD and sends it to the printer, use the `CMGetPS2ColorRenderingVMSize` function.

```
pascal CMError CMGetPS2ColorRenderingVMSize
                      (CMProfileRef srcProf, CMProfileRef dstProf,
                       unsigned long *vmSize,
                       Boolean *preferredCMMnotfound);
```

`srcProf` A profile reference to the source profile whose CMM is to be used.

`dstProf` A profile reference to the destination printer profile.

`vmSize` The virtual memory size of the CRD, returned by the function.

`preferredCMMnotfound`
A flag that returns `true` if the preferred CMM was not available or if it was unable to perform the function and the default CMM was used. Otherwise returns `false`.

DESCRIPTION

Your application or device driver can call this function to determine if the virtual memory size of the color rendering dictionary exceeds the printer's capacity before sending the CRD to the printer. If the printer's profile contains the Apple-defined optional tag `'psvm'` described in "PostScript Color Rendering Dictionary (CRD) Virtual Memory Size Tag Structure" on page 3-48, then the Apple-supplied CMM will return the data supplied by this tag specifying the CRD virtual memory size for the rendering intent's CRD. If the printer's profile does not contain this tag, then the CMM uses an algorithm to assess the VM size of the CRD, in which case the assessment may be larger than the actual maximum VM size.

The source profile specifies the rendering intent to be used.

Locating the ColorSync Profiles Folder

Your application can use this function to determine the location of the ColorSync™ Profiles folder.

CMGetColorSyncFolderSpec

To obtain the hierarchical file system (HFS) reference number and the directory ID for the ColorSync™ Profiles folder, use the `CMGetColorSyncFolderSpec` function.

```
pascal CMError CMGetColorSyncFolderSpec (short vRefNum,
                        Boolean createFolder,
                        short *foundVRefNum,
                        long *foundDirID);
```

`vRefNum` The reference number of the volume to be examined. The volume must be mounted. The constant `kOnSystemDisk` defined in the Folders header file (`Folders.h`)specifies the active system volume.

`createFolder`
A flag you set to `true` to direct the ColorSync Manager to create the ColorSync Profiles folder, if it does not exist. You can use the constants `kCreateFolder` and `kDontCreateFolder`, defined in `Folders.h` file, to assign a value to the flag.

`foundVRefNum`
The HFS volume reference number. The ColorSync Manager returns this value if the function completes successfully.

`foundDirID` The HFS directory ID. ColorSync returns this value if the function completes successfully.

DESCRIPTION

If the ColorSync Profiles folder does not already exist, you can use this function to create it.

SEE ALSO

For information about the Macintosh file system, see *Inside Macintosh: Files.*

Application-Defined Functions for the ColorSync Manager

Your application supplies the following functions for use with the ColorSync Manager API functions. The ColorSync Manager API functions that use your functions take a pointer to your function as an input parameter.

MyColorSyncDataTransfer

Your `MyColorSyncDataTransfer` function transfers profile data from the format for embedded profiles to disk file format or vice versa. The `MyColorSyncDataTransfer` function is also used by the PostScript functions to transfer data from a profile to text format usable by a PostScript driver.

This application-supplied function must conform to the following declaration. For example, this is how you should declare the function if you were to name it `MyColorSyncDataTransfer`:

```
pascal OSErr MyColorSyncDataTransfer(long command, long *size,
                                    void *data, void *refCon);
```

`command` The command with which the `MyColorSyncDataTransfer` function is called. This command specifies the operation the function is to perform.

`size` On entry, the size in bytes of the data to be transferred. On return, the size of the data actually transferred.

`data` A pointer to the buffer supplied by the ColorSync Manager to be used for the data transfer.

`refCon` A reference constant used to hold the application data passed in from the `CMFlattenProfile` function, the `CMUnflattenProfile` function, or the `CMGetPS2ColorSpace`, `CMGetPS2ColorRenderingIntent`, or `CMGetPS2ColorRendering` functions. Each time the CMM calls your `MyColorSyncDataTransfer` function, it passes this data to the function.

DESCRIPTION

Your `MyColorSyncDataTransfer` function is called to flatten and unflatten profiles or to transfer PostScript-related data from a profile to the PostScript format to be sent to an application or device driver.

The ColorSync dispatcher and the CMM communicate with the `MyColorSyncDataTransfer` function using the command parameter to identify the operation to be performed. To read and write profile data, your function must the support the following commands: `openReadSpool`, `openWriteSpool`, `readSpool`, `writeSpool`, and `closeSpool`.

You determine the behavior of your `MyColorSyncDataTransfer` function. This section describes how your function might handle the flattening and unflattening processes.

FLATTENING A PROFILE

The ColorSync Manager calls the specified profile's preferred CMM when an application calls the `CMFlattenProfile` function to transfer profile data embedded in a graphics document.

The ColorSync Manager determines if the CMM supports the `CMFlattenProfile` function. If so, the ColorSync Manager dispatches the `CMFlattenProfile` function to the CMM. If not, ColorSync calls the default Apple-supplied CMM dispatching the `CMFlattenProfile` function to it.

The CMM communicates with the `MyColorSyncDataTransfer` function using a command parameter to identify the operation to be performed. The CMM calls your function as often as necessary, passing to it on each call any data transferred to the CMM from the `CMFlattenProfile` function's `refCon` parameter.

The ColorSync Manager calls your function with the following sequence of commands: `openWriteSpool`, `writeSpool`, and `closeSpool`. Here is how you should handle these commands:

- When the CMM calls your function with the `openWriteSpool` command, you should perform any initialization required to write profile data you receive from the CMM to a buffer or file.
- The CMM will call your function with the `writeSpool` command as many times as necessary to transfer all the profile data to you. Each time you are called, you should receive the data and write it to your buffer or file, returning in the `size` parameter the number of bytes of data you actually accepted.
- When the CMM calls your function with the `closeSpool` command, you should perform any required cleanup processes.

As part of this process, your function can embed the profile data in a graphics document, for example, a PICT file or a TIFF file. For example, your `MyColorSyncDataTransfer` function can call the QuickDraw `PicComment` function to embed the flattened profile in a picture.

UNFLATTENING A PROFILE

When an application calls the `CMUnflattenProfile` function to transfer a profile that was embedded in a graphics document to an independent disk file, the ColorSync dispatcher calls your `MyColorSyncDataTransfer` function to obtain the preferred CMM of the profile to be flattened. The ColorSync Manager calls

your function with the following sequence of commands: `openReadSpool`, `readSpool`, `closeSpool`. Here is how you should handle these commands:

- When the CMM calls your function with the `openReadSpool` command, you should perform any initialization required to read from the embedded profile format.
- The CMM will call your function with the `readSpool` command as many times as necessary, directing your function to extract the profile data from the embedded format in the image file and return it to the CMM in the `data` buffer. The CMM passes in the `size` parameter the number of bytes of data you should return. Each time you are called, you should read and return the data, also returning in the `size` parameter the number of bytes of data you actually returned to the CMM.
- When the CMM calls your function with the `closeSpool` command, you should perform any required cleanup processes.

The preferred CMM is stored in the profile header's `CMMType` field. The `MyColorSyncDataTransfer` function must be able to buffer at least 8 bytes of data to hold the `CMMType` field value.

The ColorSync Manager determines if the CMM supports the `CMUnflattenProfile` function. If so, the ColorSync Manager calls the preferred CMM to dispatch the `CMUnflattenProfile` function to it. If not, the ColorSync Manager calls the default Apple-supplied CMM to dispatch the `CMUnflattenProfile` function to it.

The CMM calls the calling program-supplied `MyColorSyncDataTransfer` to direct it to unflatten the profile data and write it to a disk file. The CMM calls your function as often as necessary, passing to it on each call any data transferred to the CMM from the `CMUnflattenProfile` function's `refCon` parameter.

MyCMBitmapCallBackProc

Your `MyCMBitmapCallBackProc` function reports on the progress of a color-matching or color-checking session being performed for a bitmap or a pixel map.

This application-supplied function must conform to the following declaration. For example, this is how you should declare the function if you were to name it MyCMBitmapCallBackProc:

```
pascal Boolean MyCMBitmapCallBackProc (long progress,
                                                           void *refCon);
```

progress A byte count that begins at an arbitrary value and counts down to one when the matching is complete.

refCon The reference constant passed to your MyCMBitmapCallBackProc function each time the Color Management Module (CMM) calls your function.

DESCRIPTION

Your MyCMBitmapCallBackProc function allows your application to monitor the progress of a color-matching or color-checking session for a bitmap or a pixel map. Your function can also terminate the matching or checking operation.

Your callback function is called by the CMM performing the matching or checking process if your application passes a pointer to your callback function in the progressProc parameter when it calls one of the following functions: CWMatchPixMap described on page 3-88, CWCheckPixMap described on page 3-90, CWMatchBitmap described on page 3-92, and CWCheckBitMap described on page 3-95.

The CMM used for the session calls your function at regular intervals. For example, the Apple-supplied CMM calls your function approximately every half second unless the color matching or checking occurs quickly enough to warrant not calling your function; this happens when there is a small amount of data to be matched or checked.

Each time the ColorSync Manager calls your function, it passes to the function any data stored in the reference constant. This is the data that your application specified in the refCon parameter when it called one of the color matching or checking functions.

For large bitmaps and pixel maps, your application could display some type of indicator to users, such as a progress thermometer, to register how much has been done and how much is yet to be done. If your application were to do this, for example, you could use the reference constant to pass the dialog box's window reference to the callback function.

To terminate the matching or checking operation, your function should return a value of `true`. Because pixel map matching is done in place, an application that allows the user to terminate the process should revert to the prematched image to avoid partial mapping.

For bitmap matching, if the `matchedBitMap` parameter of the `CWMatchBitmap` function specifies `NULL` to indicate that the source bitmap is to be matched in place and the application allows the user to abort the process, you should also revert to the prematched bitmap if the user terminates the operation.

Each time the ColorSync Manager calls your progress function, it passes a byte count in the `progress` parameter. The last time the ColorSync Manager calls your progress function, it passes a byte count of zero to indicate the completion of the matching or checking process. You should use the zero byte count as a signal to perform any cleanup operations your function requires, such as filling the thermometer or progress bar to completion to indicate to the user the end of the checking or matching session, and then removing the dialog box used for the display.

MyCMProfileFilterProc

After a profile has been included in the profile search result based on criteria specified in the search record, your `MyCMProfileFilterProc` function can examine the profile whose reference you specify to further determine if it should be included or excluded from the profile search result list based on criteria such as an element or elements not included in the `CMSearchRecord` search record. Your `MyCMProfileFilterProc` function can also perform searching using `AND` or `OR` logic.

This application-supplied function must conform to the following declaration. For example, this is how you should declare the function if you were to name it `MyCMProfileFilterProc`:

```
pascal Boolean MyCMProfileFilterProc (CMProfileRef prof,
                        void *refCon);
```

`prof` A reference to the profile to be tested for filtering.

`refCon` A reference constant that holds data passed through from the `CMNewProfileSearch` function or the `CMUpdateProfileSearch` function.

DESCRIPTION

Your `MyCMProfileFilterProc` function is called after the `CMNewProfileSearch` function searches for profiles based on the search record's contents as specified by the search bitmask.

When your application calls the `CMNewProfileSearch`, it gives a reference to a search specification record of type `CMSearchRecord`, described on page 3-33, that contains a `filter` field. If the `filter` field contains a pointer to your `MyCMProfileFilterProc` function, then your function is called to determine if a profile is to be eliminated from the search result list. Your function should return `true` for a given profile to exclude that profile from the search result list. If you do not want to filter profiles beyond the criteria in the search record, specify a `NULL` value for the search record's `filter` field.

Result Codes

noErr	0	No error
cmProfileError	–170	There is something wrong with the content of the profile
cmMethodError	–171	An error occurred during the CMM arbitration process that determines the CMM to be used
cmMethodNotFound	–175	CMM not present
cmProfileNotFound	–176	Responder error
cmProfilesIdentical	–177	Profiles are the same
cmCantConcatenateError	–178	Profiles can't be concatenated
cmCantXYZ	–179	CMM does not handle XYZ color space
cmCantDeleteProfile	–180	Responder error
cmUnsupportedDataType	–181	Responder error
cmNoCurrentProfile	–182	Responder error
cmElementTagNotFound	–4200	The tag you specified is not in the specified profile
cmIndexRangeErr	–4201	Index out of range
cmFatalProfileErr	–4203	Returned from File Manager while updating a profile file in response to CMUpdateProfile Profile content may be corrupted
cmInvalidProfileLocation	–4205	Operation not supported for this profile location
cmInvalidSearch	–4206	Bad search handle
cmSearchError	–4207	Internal error occurred during profile search.
cmInvalidColorSpace	–4209	Profile color space does not match bitmap type
cmInvalidSrcMap	–4210	Source pixel map or bitmap was invalid
cmInvalidDstMap	–4211	Destination pix/bit map was invalid
cmNoGDevicesError	–4212	Begin matching or ending matching—no gdevices available
cmInvalidProfileComment	–4213	Bad profile comment during drawpicture
cmRangeoverFlow	–4214	One or more output color value overflows in color conversion: all input color values will be converted and the overflow will be clipped
cmCantCopyModifiedV1Profile	–4215	It is illegal to copy version 1.0 profiles that have been modified

ColorSync Manager Reference for Color Management Modules

Contents

Constants 4-3
Color Management Module Component Interface 4-3
Required Request Codes 4-4
Optional Request Codes 4-5
Required Functions 4-9
MyNCMInit 4-9
MyCMMatchColors 4-11
MyCMCheckColors 4-12
Optional Functions 4-14
MyCMMValidateProfile 4-15
MyCMMatchBitmap 4-16
MyCMCheckBitmap 4-19
MyCMConcatInit 4-22
MyCMMatchPixMap 4-24
MyCMCheckPixMap 4-27
MyCMNewLinkProfile 4-29
MyCMMGetPS2ColorSpace 4-31
MyCMMGetPS2ColorRenderingIntent 4-33
MyCMMGetPS2ColorRendering 4-35
MyCMMGetPS2ColorRenderingVMSize 4-38
MyCMMFlattenProfile 4-40
MyCMMUnflattenProfile 4-42

This document was created with FrameMaker 4.0.4

This chapter describes the request code constants passed to your color management module (CMM) from the Component Manager when a ColorSync-supportive application or device driver calls a ColorSync Manager function to request services your CMM provides.

Your CMM must support a required subset of these request codes, and it should support the others. This chapter also describes the functions your CMM may define to respond to these ColorSync Manager request codes. For information describing how to develop a CMM that responds to the ColorSync Manager request codes, see the chapter "Developing Color Management Modules" in *Advanced Color Imaging on the Mac OS.*

Constants

This section describes the constants for the CMM component interface version and the ColorSync Manager request codes.

Color Management Module Component Interface

If your CMM supports the ColorSync Manager version 2.0, it should return the constant defined by the following enumeration when the Component Manager calls your CMM with the `kComponentVersionSelect` request code:

```
enum {
    CMMInterfaceVersion = 1
    };
```

In response to the `kComponentVersionSelect` request code, a CMM should set its entry point function's result to the CMM version number. The high-order 16 bits represent the major version and the low-order 16 bits represent the minor version. The `CMMInterfaceVersion` constant represents the major version number.

Note
A CMM that supports ColorSync 1.0 returns 0 for the major version in response to the version request. ◆

The `kComponentVersionSelect` request code is one of four required Component Manager requests your CMM must handle. For complete details on the

This document was created with FrameMaker 4.0.4

Component Manager required request codes, see the chapter “Component Manager” in *Inside Macintosh: More Macintosh Toolbox.*

Required Request Codes

Your CMM must respond to the ColorSync Manager required request codes. When a CMM receives a required request code from the ColorSync Manager, the CMM must 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. For a description of how your CMM can respond to ColorSync Manager requests from the Component Manager, see the chapter “Developing Color Management Modules” in *Advanced Color Imaging on the Mac OS.*

The ColorSync Manager defines the following required request codes:

```
enum {
    kCMMInit            = 0,
    kCMMMatchColors     = 1,
    kCMMCheckColors     = 2
    kNCMMInit           = 6,
};
```

Constant descriptions

`kCMMInit` This request code is provided for backward compatibility with ColorSync 1.0. A CMM that supports ColorSync 1.0 profiles should respond to this request code by initializing any private data required for the color-matching or gamut-checking session to be held as indicated by subsequent request codes. If your CMM supports only ColorSync 1.0 profiles or both ColorSync 1.0 profiles and ColorSync Manager version 2.0 profiles, you must support this request code. If you support only ColorSync Manager version 2.0 profiles, you should return an unimplemented error in response to this request code.

`kCMMMatchColors` In response to this request code, your CMM should match the colors in the `myColors` parameter to the color gamut of the destination profile and replace the color-list color values with the matched colors. For more information

about how your CMM should respond to this request code, see the `MyCMMatchColors` function on page 4-11.

`kCMMCheckColors` In response to this request code, your CMM should test the given list of colors in the `myColors` parameter against the gamut specified by the destination profile and report if the colors fall within a destination device's color gamut. For more information about how your CMM should respond to this request code, see the `MyCMCheckColors` function on page 4-12.

`kNCMMInit` In response to this request code, your CMM should initialize any private data it will need for the color session and for subsequent requests from the calling application or driver. For more information about how your CMM should respond to this request code, see the `MyNCMInit` function on page 4-9.

Optional Request Codes

Your CMM should respond to the ColorSync Manager request codes defined by the following enumeration, but it is not required to do so. For a description of how your CMM can respond to ColorSync Manager requests from the Component Manager, see "Developing Color Management Modules" in *Advanced Color Imaging on the Mac OS.*

The ColorSync Manager defines the following optional request codes:

```
enum {
    kCMMMatchPixMap                 = 3,
    kCMMCheckPixMap                 = 4,
    kCMMConcatenateProfiles         = 5,
    kCMMConcatInit                  = 7,
    kCMMValidateProfile             = 8,
    kCMMMatchBitmap                 = 9,
    kCMMCheckBitmap                 = 10,
    kCMMGetPS2ColorSpace            = 11,
    kCMMGetPS2ColorRenderingIntent  = 12,
    kCMMGetPS2ColorRendering        = 13,
    kCMMFlattenProfile              = 14,
    kCMMUnflattenProfile            = 15,
```

```
kCMMNewLinkProfile             = 16,
kCMMGetPS2ColorRenderingVMSize = 17
};
```

Constant descriptions

`kCMMMatchPixMap` In response to this request code, your CMM must match the colors of the pixel map image pointed to by the `myPixMap` parameter to the gamut of the destination device, replacing the original pixel colors with their corresponding colors as specified in the data color space of the destination device's color gamut. To perform the matching, you use the profiles specified by a previous `kNCMMInit`, `kCMMInit`, or `kCMMConcatInit` request to your CMM. For more information about how your CMM should respond to this request code, see the `MyCMMatchPixMap` function on page 4-24.

`kCMMCheckPixMap` In response to this request code, your CMM must check the colors of the pixel map image pointed to by the `myPixMap` parameter against the gamut of the destination device to determine if the pixel colors are within the gamut of the destination device and report the results. To perform the check, you use the profiles specified by a previous `kNCMMInit`, `kCMMInit`, or `kCMMConcatInit` request to your CMM. For more information about how your CMM should respond to this request code, see the `MyCMCheckPixMap` function on page 4-27.

`kCMMConcatenateProfiles`
This request code is for backward compatibility with ColorSync 1.0.

`kCMMConcatInit` In response to this request code, your CMM should 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. Your function should also initialize any additional private data needed in handling subsequent calls pertaining to this component instance. For more information about how your CMM should respond to this request code, see the `MyCMConcatInit` function on page 4-22.

`kCMMValidateProfile`
In response to this request code, your CMM should test the

profile whose reference is passed in the `prof` parameter to determine if the profile contains the minimum set of elements required for a profile of its type. For more information about how your CMM should respond to this request code, see the `MyCMMValidateProfile` function on page 4-15.

`kCMMMatchBitmap` In response to this request code, your CMM must match the colors of the source image bitmap pointed to by the `bitmap` parameter to the gamut of the destination device using the profiles specified by a previous `kNCMMInit`, `kCMMInit`, or `kCMMConcatInit` request to your CMM. For more information about how your CMM should respond to this request code, see the `MyCMMatchBitmap` function on page 4-16.

`kCMMCheckBitmap` In response to this request code, your CMM must check the colors of the source image bitmap pointed to by the `bitmap` parameter against the gamut of the destination device using the profiles specified by a previous `kNCMMInit`, `kCMMInit`, or `kCMMConcatInit` request to your CMM. For more information about how your CMM should respond to this request code, see the `MyCMCheckBitmap` function on page 4-19.

`kCMMGetPS2ColorSpace`
In response to this request code, your CMM must obtain or derive the color space element data from the source profile whose reference is passed to your function in the `srcProf` parameter and pass the data to a low-level data-transfer function supplied by the calling application or device driver. For more information about how your CMM should respond to this request code, see the `MyCMMGetPS2ColorSpace` function on page 4-31.

`kCMMGetPS2ColorRenderingIntent`
In response to this request code, your CMM must obtain the color-rendering intent from the header of the source profile whose reference is passed to your function in the `srcProf` parameter and then pass the data to a low-level data-transfer function supplied by the calling application or device driver. For more information about how your CMM should respond to this request code, see the `MyCMMGetPS2ColorRenderingIntent` function on page 4-33.

`kCMMGetPS2ColorRendering`
: In response to this request code, your CMM must obtain the rendering intent from the source profile's header and 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 about how your CMM should respond to this request code, see the `MyCMMGetPS2ColorRendering` function on page 4-35.

`kCMMFlattenProfile`
: In response to this request code, your CMM must extract the profile data from the profile to be flattened, identified by the `prof` parameter, and pass the profile data to the function specified in the `proc` parameter. For more information about how your CMM should respond to this request code, see the `MyCMMFlattenProfile` function on page 4-40.

`kCMMUnflattenProfile`
: In response to this request code, your CMM must create a temporary file in which to store the profile data you receive from the low-level data-transfer function supplied by the calling application or driver. Your function must return the file specification. For more information about how your CMM should respond to this request code, see the `MyCMMUnflattenProfile` function on page 4-42.

`kCMMNewLinkProfile`
: In response to this request code, your CMM 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. For more information about how your CMM should respond to this request code, see the `MyCMNewLinkProfile` function on page 4-29.

`kCMMGetPS2ColorRenderingVMSize`
: In response to this request code, your CMM must obtain or assess the maximum virtual memory (VM) size of the CRD specified by the destination profile. The CRD whose size you return must be that of the dictionary for the rendering intent specified by the source profile. For more information about how your CMM should respond to this request

code, see the `MyCMMGetPS2ColorRenderingVMSize` function on page 4-38.

Required Functions

This section describes the functions that your CMM should define to handle ColorSync Manager required request codes.

MyNCMInit

A CMM must respond to the `kNCMMInit` request code. The ColorSync Manager sends this code to request your CMM to instantiate any private data it needs. A CMM responds to the `kNCMMInit` request code by calling a CMM-defined subroutine, for example, `MyNCMInit` to handle the request.

```
pascal CMError MyNCMInit (ComponentInstance CMSession, CMProfileRef
                          srcProfile, CMProfileRef dstProfile);
```

`CMSession` A handle to your CMM's storage for the instance of your component associated with the calling application or device driver.

`srcProfile` A reference to the source profile to be used in the color-matching or color-checking session to be set up. Your CMM should store any profile information it requires before returning to the Component Manager. (The calling program obtained the profile reference passed in this parameter.)

`dstProfile` A reference to the destination profile to be used in the color-matching or color-checking session to be set up. Your CMM should store any profile information it requires before returning to the Component Manager. (The calling program obtained the profile reference passed in this parameter.)

DESCRIPTION

The Component Manager calls your CMM with the `kNCMMInit` request code when a ColorSync-supportive application or device driver specifies your CMM

for a color-matching or color-checking session. For example, when an application or device driver calls the `NCWNewColorWorld` function, the Component Manager calls your `MyNCMInit` function.

Using the storage pointed to by the `CMSession` handle, your `MyNCMInit` function should initialize any private data your CMM will need for the color session and for handling subsequent calls pertaining to this component instance. Your function must obtain required information from the profiles and initialize private data for subsequent color-matching or color-checking sessions with these values. After your function returns to the Component Manager, it no longer has access to the profiles.

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.

In addition to the standard profile information you should preserve in response to this request, you should preserve the quality flag setting specified in the profile header and the rendering intent, also specified in the header.

The Component Manager calls your CMM with a standard open request to open the CMM when a ColorSync-supportive application or device driver requests that the Component Manager open a connection to your component. At this time, your component should allocate any memory it needs in order to maintain a connection for the requesting application or driver. You should allocate memory from the current heap zone. It 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. Whenever the calling application or driver requests services from your component, the Component Manager supplies you with the handle to this memory in the `CMSession` parameter.

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” in the *Advanced Color Imaging Reference* on the enclosed CD.

MyCMMatchColors

A CMM must respond to the `kCMMMatchColors` request code. The ColorSync Manager sends this request code to your CMM on behalf of an application or device driver that called the `CWMatchColors` function or high-level QuickDraw operations.

The ColorSync Manager dispatches this request to the Component Manager, which calls your CMM to service the request. A CMM typically responds to the `kCMMMatchColors` request code by calling a CMM-defined function (for example, `MyCMMatchColors`) to handle the request by matching colors in the color list.

```
pascal CMError MyCMMatchColors (ComponentInstance CMSession,
                         CMColor *myColors,
                         unsigned long count);
```

`CMSession` A handle to your CMM's storage for the instance of your component associated with the calling application or device driver.

`myColors` A pointer to an array of type `CMColor` specified by the calling application or device driver. On entry, this array contains the list of colors to be matched. The color values are given in the data color space of the source profile specified by a previous `kNCMMInit` or `kCMMConcatInit` request to your CMM. On return, this array contains the list of matched colors specified by your function in the data color space of the destination profile. For a description of the `CMColor` data type, see the description of the section "The Color Union" in the chapter "ColorSync Manager Reference for Applications and Device Drivers" on the enclosed CD.

`count` A one-based count of the number of colors in the color list of the `CMColor` array.

DESCRIPTION

Before the Component Manager calls your CMM with a ColorSync request to match colors, it calls your CMM with a `kNCMMInit`, `kCMMInit`, or `kCMMConcatInit` request passing your CMM references to the profiles to be used for the color matching-session and requesting your CMM to initialize the session.

If the Component Manager calls your CMM with a ColorSync `kNCMMInit` or `kCMMInit` request code, it passes references to the source and destination profiles to be used for the color-matching session. If it calls your CMM with the ColorSync `kCMMConcatInit` request code, it passes a pointer to an array of type `ConcatProfileSet` containing a set of profiles or a device-linked profile specified by the calling application to be used for the color-matching session. For information about the `ConcatProfileSet` data type, see the section "Concatenated Profile Set Structure" in "ColorSync Manager Reference for Applications and Device Drivers" in the *Advanced Color Imaging Reference* on the enclosed CD. This chapter also explains the rules governing concatenated profiles and device-linked profiles.

When the Component Manager calls your CMM with the `kCMMMatchColors` request code, it passes to your CMM in the `CMSession` parameter a handle to your CMM's storage for the calling applications's component instance.

In response to this request code, you must support 16-bit components for color spaces other than multichannel components and 8-bit components for HiFi colors.

Using the profile data you set in your storage for this component instance, your `MyCMMatchColors` function should match the colors specified in the `myColors` array to the color gamut of the destination profile, replacing the color value specifications in the `myColors` array with the matched colors specified in the data color space of the destination profile. If you used some other method to store profile data for this component instance when you initialized the session, you should obtain the profile data you require for the color matching from that storage. The color list may contain multichannel color data types, so your CMM must support them.

MyCMCheckColors

A CMM must respond to the `kCMMCheckColors` request code. The ColorSync Manager sends this request code to your CMM on behalf of an application or device driver that called the `CWCheckColors` function. The ColorSync Manager dispatches this request to the Component Manager, which calls your CMM to service the request. A CMM typically responds to the `kCMMCheckColors` request

code by calling a CMM-defined function (for example, `MyCMCheckColors`) to handle the request.

```
pascal CMError MyCMCheckColors (ComponentInstance CMSession,
                    CMColor *myColors,
                    unsigned long count,
                    long *result);
```

`CMSession` A handle to your CMM's storage for the instance of your component associated with the calling application or device driver.

`myColors` A pointer to an array of type `CMColor` specified by the calling application or device driver. On entry, this array contains the list of colors to be checked against the destination device's color gamut. The color values are given in the data color space of the source profile specified by a previous `kNCMMInit` or `kCMMConcatInit` request to your CMM. For a description of the `CMColor` data type, see the description of the section "The Color Union" in the chapter "ColorSync Manager Reference for Applications and Device Drivers" in the *Advanced Color Imaging Reference* on the enclosed CD.

`count` A one-based count of the number of colors in the color list of the `CMColor` array.

`result` A pointer to an array of long data types used as a bitfield, with each bit representing a color in the array pointed to by `myColors`. The `result` array contains enough members to allow for 1 bit to represent each color in the `myColors` array. Your function sets a bit in the array if the corresponding color-list color is out of gamut for the destination profile. On return, this array indicates the color-checking results.

DESCRIPTION

When your CMM receives a `kCMMCheckColors` request code, 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. Before the Component Manager calls your CMM with a ColorSync request to gamut check colors, it calls your CMM with a `kNCMMInit`, `kCMMInit`, or `kCMMConcatInit` request passing your CMM references to the profiles to be used

for the color-checking session and requesting your CMM to initialize the session.

If the Component Manager calls your CMM with a ColorSync `kNCMMInit` or `kCMMInit` request, it passes references to the source and destination profiles to be used for the color-checking session. (If it calls your CMM with the ColorSync `kCMMConcatInit` request, it passes a pointer to an array of type `ConcatProfileSet` containing a set of profiles or a device-linked profile specified by the calling program to be used for the color-checking session.)

When the Component Manager calls your CMM with the `kCMMCheckColors` request code, it passes to your CMM in the `CMSession` parameter a handle to your CMM's storage for the calling application's or device driver's component instance. This is the storage whose data you initialized when the Component Manager called you to initialize the session for this component instance.

Using the profile data set in your storage for this component instance, your `MyCMCheckColors` function should check the colors specified in the `myColors` array against the color gamut of the destination profile. Your function should use the `result` array to return indication of whether the colors in the list are in or out of gamut for the destination device. If you used some other method to store profile data for this component instance when you initialized the session, you should obtain the profile data you require for the color matching from that storage. The color list may contain multichannel color data types, so your CMM must support them. (If your CMM does not support them, you should return an unimplemented error in response to the initialization request code. See `MyNCMInit`, beginning on page 4-9, and `MyCMConcatInit`, beginning on page 4-22, for more information.)

For each color in the list, your `MyCMCheckColors` function should set the corresponding bit in the `result` bit array if the color is out of gamut for the destination device as specified by the destination profile. The leftmost bit in the field corresponds to the first color in the list.

The gamut test your function performs provides a preview of color matching. The ColorSync Manager returns the results to the calling application or device driver.

Optional Functions

This section describes the functions that your CMM should define to handle ColorSync Manager optional request codes.

MyCMMValidateProfile

A CMM should respond to the `kCMMValidateProfile` request code, but it is not required to do so. The ColorSync Manager sends this request code to your CMM on behalf of an application or device driver that called the `CMValidateProfile` function. The ColorSync Manager dispatches this request to the Component Manager, which calls your CMM to service the request. A CMM typically responds to the `kCMMValidateProfile` request code by calling a CMM-defined function (for example, `MyCMMValidateProfile`) to handle the request.

```
pascal CMError MyCMMValidateProfile (ComponentInstance CMSession,
                        CMProfileRef prof,
                        Boolean *valid);
```

`CMSession` A handle to your CMM's storage for the instance of your component associated with the calling application or device driver.

`prof` A reference to the profile to be tested for required profile elements.

`valid` A flag whose value you set to `true` if the profile contains the elements required for a color-matching or color-checking session for a profile of this type and `false` if it doesn't.

DESCRIPTION

Your `MyCMMValidateProfile` function should test the profile whose reference is passed in the `prof` parameter to determine if the profile contains the minimum set of elements required for a profile of its type. For each profile type, such as a device profile, there is a specific set of required tagged elements defined by the ICC that the profile must include.

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.

Your `MyCMMValidateProfile` function should check for the existence of the required minimum set of profile elements for a profile of this type and any optional or private tags required by your CMM.

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. The signature of the Apple-supplied CMM is `'appl'`.

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, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox.*

MyCMMatchBitmap

A CMM should respond to the `kCMMMatchBitmap` request code, but it is not required to do so. The ColorSync Manager sends this request code to your CMM on behalf of an application or device driver that called the `CWMatchBitMap` function or high-level QuickDraw operations. The ColorSync Manager dispatches this request to the Component Manager, which calls your CMM to service the request. A CMM typically responds to the `kCMMMatchBitmap` request code by calling a CMM-defined function (for example, `MyCMMatchBitmap`) to handle the request.

```
pascal CMError MyCMMatchBitmap(ComponentInstance CMSession,
                        const CMBitmap *bitmap,
                        CMBitmapCallBackUPP progressProc,
                        void *refCon,
                        CMBitmap *matchedBitmap);
```

`CMSession` A handle to your CMM's storage for the instance of your component associated with the calling application or device driver.

`bitmap` A pointer to the bitmap containing the source image data whose colors your function must match.

`progressProc` A pointer to a callback function supplied by the calling application or device driver that monitors the color-matching progress or aborts the operation as your function matches the bitmap colors. Your `MyCMMatchBitmap` function must call this function periodically to allow it to report progress to the user.

`refCon` A reference constant passed from the calling application or driver, which your `MyCMMatchBitmap` function must pass through as a parameter to calls it makes to the `MyCMBitmapCallBackProc` function.

`matchedBitMap` A pointer to a bitmap in which your function stores the resulting color-matched image. The calling program allocates the pixel buffer pointed to by the `image` field of the `CMBitMap` structure. If this value is `null`, then your `MyCMMatchBitmap` function must match the bitmap colors in place.

DESCRIPTION

If your CMM supports this request code, your `MyCMMatchBitmap` function should be prepared to receive any of the bitmap types defined by the ColorSync Manager. Your `MyCMMatchBitmap` function must match the colors of the source image bitmap pointed to by `bitmap` to the color gamut of the destination profile using the profiles specified by a previous `kNCMMInit`, `kCMMInit`, or `kCMMConcatInit` request to your CMM. If the `matchedBitMap` parameter points to a bitmap, you should store the resulting color-matched image in that bitmap. Otherwise, you should store the resulting color-matched image in the source bitmap pointed to by the `bitmap` parameter. The color-matched bitmap image your function creates is returned to the calling application or driver.

Before the Component Manager calls your CMM with a ColorSync request to match the colors of a bitmap, it calls your CMM with a `kNCMMInit`, `kCMMInit`, or `kCMMConcatInit` request passing your CMM references to the profiles to be used for the color matching session and requesting your CMM to initialize the session.

If the Component Manager calls your CMM with a ColorSync `kNCMMInit` or `kCMMInit` request, it passes references to the source and destination profiles to be used for the color-matching session. If it calls your CMM with the ColorSync `kCMMConcatInit` request code, it passes a pointer to an array of type `ConcatProfileSet` containing a set of profiles or a device-linked profile specified by the calling program to be used for the color-matching session. For information about the `ConcatProfileSet` data type, see the section

"Concatenated Profile Set Structure" in "ColorSync Manager Reference for Applications and Device Drivers" in the *Advanced Color Imaging Reference* on the enclosed CD.

When the Component Manager calls your CMM with the kCMMMatchColors request code, it passes to your CMM in the CMSession parameter a handle to your CMM's storage for the calling applications's component instance. Your MyCMMatchBitmap function should use the profile data you set in your storage for this component instance to perform the color matching. If you used some other method to store profile data for this component instance when you initialized the session, you should obtain the profile data you require for the color matching from that storage.

Your MyCMMatchBitmap function must call the progress function supplied by the calling application or device driver at regular intervals to allow it to report progress to the user on the color-matching session. 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. The Apple-supplied CMM calls the MyCMBitmapCallBackProc function approximately every half second unless color matching occurs quickly enough to warrant not calling it at all; this happens when there is a small amount of data to be matched.

Here is the prototype for the MyCMBitmapCallBackProc function pointed to by the progressProc parameter:

```
pascal Boolean MyCMBitmapCallBackProc (long progress,
                                       void *refCon);
```

Each time your MyCMMatchBitmap function calls the MyCMBitmapCallBackProc function, it must pass to the function any data stored in the reference constant. When the Component Manager calls your CMM with the kCMMMatchBitmap request code, it passes to your CMM the reference constant from the calling program.

Each time your function calls the MyCMBitmapCallBackProc function, your function must pass it a byte count in the progress parameter identifying the remaining number of bytes. The last time your MyCMMatchBitmap function calls the MyCMBitmapCallBackProc function, it must pass a byte count of 0. A byte count of 0—meaning there is no more data to match—indicates the completion of the matching process and signals the progress function to perform any cleanup operations it requires.

If the source profile's `dataColorSpace` field value and the `space` field value of the source bitmap pointed to by the `bitMap` parameter do not specify the same data color space, your function should terminate the color-matching process and return an error code.

Also, if the destination profile's `dataColorSpace` field value and the `space` field value of the resulting bitmap pointed to by the `matchedBitMap` parameter do not specify the same data color space, your function should terminate the color-matching process and return an error code.

If your CMM does not support a bitmap type that you receive, you can return an unimplemented error. In this case, the ColorSync Manager unpacks the colors of the bitmap and calls your `CMMMatchColors` function, passing it the bitmap colors in a color list. You should avoid defaulting to this behavior, if possible, because it incurs overhead and slows down performance.

MyCMCheckBitmap

A CMM should respond to the `kCMMCheckBitmap` request code, but it is not required to do so. The ColorSync Manager sends this request code to your CMM on behalf of an application or device driver that called the `CWCheckBitMap` function. The ColorSync Manager dispatches this request to the Component Manager, which calls your CMM to service the request. A CMM typically responds to the `kCMMCheckBitmap` request code by calling a CMM-defined function (for example, `MyCMCheckBitmap`) to handle the request.

```
pascal CMError MyCMCheckBitmap(ComponentInstance CMSession,
                      const CMBitmap *bitmap,
                      CMBitmapCallBackUPP progressProc,
                      void *refCon,
                      CMBitmap *resultBitmap);
```

`CMSession` A handle to your CMM's storage for the instance of your component associated with the calling application or device driver.

`bitmap` A pointer to the bitmap containing the source image data whose colors your function must check.

`progressProc` A pointer to a callback function supplied by the calling application or device driver that monitors the color-checking progress or aborts the operation as your function checks the colors of the source image. Your `MyCMCheckBitmap` function must call this function periodically to allow it to report progress to the user.

`refCon` A reference constant passed from the calling application or driver, which your `MyCMCheckBitmap` function must pass through as a parameter to calls it makes to the `MyCMBitmapCallBackProc` function.

`resultBitmap` A pointer to the resulting bitmap allocated by the calling application or device driver. Your `MyCMCheckBitmap` function must set pixels of the bitmap image to 1 if the corresponding pixel of the source bitmap indicated by `bitMap` is out of gamut.

DESCRIPTION

If your CMM supports this request code, your `MyCMMCheckBitmap` function should be prepared to receive any of the bitmap types defined by the ColorSync Manager. Your `MyCMCheckBitmap` function must check the colors of the source image bitmap pointed to by `bitmap` against the color gamut of the destination profile using the profiles specified by a previous `kNCMMInit`, `kCMMInit`, or `kCMMConcatInit` request to your CMM. If a pixel is out of the destination profile's color gamut, your function should set the corresponding pixel in the image of the bitmap pointed to by the `resultBitmap` parameter. The ColorSync Manager returns the resulting bitmap to the calling application or driver to report the outcome of the gamut check.

Before the Component Manager calls your CMM with a ColorSync request to gamut check the colors of a bitmap, it calls your CMM with a `kNCMMInit`, `kCMMInit`, or `kCMMConcatInit` request passing your CMM references to the profiles to be used for the color-checking session and requesting your CMM to initialize the session.

If the Component Manager calls your CMM with a ColorSync `kNCMMInit` or `kCMMInit` request, it passes references to the source and destination profiles to be used for the session. If it calls your CMM with the ColorSync `kCMMConcatInit` request code, it passes a pointer to an array of type `ConcatProfileSet` containing a set of profiles specified by the calling application to be used for the session. For information about the `ConcatProfileSet` data type, see the section "Concatenated Profile Set Structure" in the chapter "ColorSync

Manager Reference for Applications and Device Drivers" in the *Advanced Color Imaging Reference* on the enclosed CD.

When the Component Manager calls your CMM with the `kCMMMatchColors` request code, it passes to your CMM in the `CMSession` parameter a handle to your CMM's storage for the calling applications's component instance. Your `MyCMCheckBitmap` function should use the profile data you set in your storage for this component instance to perform the color-checking process. If you used some other method to store profile data for this component instance when you initialized the session, you should obtain the profile data you require for the color-checking process from that storage.

Your `MyCMCheckBitmap` function must call the progress function supplied by the calling application or device driver at regular intervals to allow it to report progress to the user on the color-checking session. Your `MyCMCheckBitmap` 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.

The Apple-supplied CMM calls the `MyCMBitmapCallBackProc` function approximately every half second unless the gamut checking occurs quickly enough to warrant not calling it at all; this happens when there is a small amount of data to be checked.

Here is the prototype for the `MyCMBitmapCallBackProc` function pointed to by the `progressProc` parameter:

```
pascal Boolean MyCMBitmapCallBackProc (long progress,
                                        void *refCon);
```

Each time your `MyCMCheckBitmap` function calls the `MyCMBitmapCallBackProc` function, it must pass to the function any data stored in the reference constant. When the Component Manager called your CMM with the `kCMMCheckBitmap` request code, it passed to your CMM the reference constant from the calling program.

Each time your function calls the `MyCMBitmapCallBackProc` function, your function must pass it a byte count in the `progress` parameter identifying the remaining number of bytes to be checked. The last time your `MyCMMatchBitmap` function calls the `MyCMBitmapCallBackProc` function, it must pass a byte count of 0 to indicate the completion of the color-checking process. This signals the progress function to perform any cleanup operations it requires.

If the source profile's `dataColorSpace` field value and the `space` field value of the source bitmap pointed to by the `bitMap` parameter do not specify the same data color space, your function should terminate the color-checking process and return an error code.

If your CMM does not support a bitmap type that you receive, you can return an unimplemented error. In this case, the ColorSync Manager unpacks the colors of the bitmap and calls your `MyCMMatchColors` function, passing it the bitmap colors in a color list. You should avoid defaulting to this behavior, if possible, because it incurs overhead and slows down performance.

MyCMConcatInit

A CMM should respond to the `kCMMConcatInit` request code, but it is not required to do so. The ColorSync Manager sends this request code to your CMM on behalf of an application or device driver that called the `CWConcatColorWorld` function. The ColorSync Manager dispatches this request to the Component Manager, which calls your CMM to service the request. A CMM typically responds to the `kCMMConcatInit` request code by calling a CMM-defined function (for example, `MyCMConcatInit`) to handle the request.

```
pascal CMError MyCMConcatInit (ComponentInstance CMSession,
                              CMConcatProfileSet *profileSet);
```

`session` A handle to your CMM's storage for the instance of your component associated with the calling application or device driver.

`profileSet` A pointer to a data structure of type `CMConcatProfileSet` containing an array of profiles to be used in a 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. For a description of the `CMConcatProfileSet` data structure, see "Concatenated Profile Set Structure" in "ColorSync Manager Reference for Applications and Device Drivers" on the enclosed CD.

DESCRIPTION

Using the private storage pointed to by the `CMSession` handle, your `MyCMConcatInit` function should 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. Your function should also initialize any additional private data needed in handling subsequent calls pertaining to this component instance.

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 function must obtain required information from the profiles and initialize private data for subsequent color-matching or color-checking session with these values. After your function returns to the Component Manager, it no longer has access to the profiles.

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.

When your CMM uses a device-linked profile or a set of concatenated profiles, you must adhere to the following guidelines and rules:

- You should use the quality flag setting—indicating normal mode, draft mode, or best mode—specified by the first profile for the entire color-matching session; you should ignore the quality flags of following profiles in the sequence. The profile header `flag` field holds the quality flag setting. Your CMM may choose to ignore the quality flag. This is allowed, but not recommended unless you support best mode by default.
- You must use the rendering intent specified by the first profile to color match to the second profile, the rendering intent specified by the second profile to color match to the third profile, and so on through the series of concatenated profiles.
- If the calling application or driver passed a color space profile in the middle of the profile sequence, the Apple-supplied CMM ignores this profile. Your CMM should also ignore it.

For specific guidelines on handling device-linked profiles and additional information on handling concatenated profiles, see the chapter “ColorSync Manager Reference for Applications and Device Drivers” in the *Advanced Color Imaging Reference* on the enclosed CD.

The Component Manager calls your CMM with a standard open request to open the CMM when a ColorSync-supportive application or device driver requests that the Component Manager open a connection to your component. At this time, your component should allocate any memory it needs in order to maintain a connection for the requesting application or driver. You should attempt to allocate memory from the current heap zone. It 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. Whenever the calling application or driver requests services from your component, the Component Manager supplies you with the handle to this memory in the `session` parameter. For complete details on the `SetComponentInstanceStorage` function, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox.*

The Component Manager may call your CMM with the `kCMMConcatInit` 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" in the *Advanced Color Imaging Reference* on the enclosed CD.

MyCMMatchPixMap

A CMM should respond to the `kCMMMatchPixMap` request code, but it is not required to do so. The ColorSync Manager sends this request code to your CMM on behalf of an application that called the `CWMatchPixMap` function or high-level QuickDraw operations. The ColorSync Manager dispatches this request to the Component Manager, which calls your CMM to service the request. A CMM typically responds to the `kCMMMatchPixMap` request code by calling a CMM-defined function (for example, `MyCMMatchPixMap`) to handle the request.

```
pascal CMError MyCMMatchPixMap(ComponentInstance CMSession,
                    PixMap *myPixMap,
                    CMBitmapCallBackUPP progressProc,
                    void *refCon);
```

`session` A handle to your CMM's storage for the instance of your component associated with the calling application or device driver.

`myPixMap` A pointer to the pixel map to be matched. A pixel map is a QuickDraw structure describing pixel data. The pixel map is stored in nonrelocatable memory. Your function replaces the original colors of the pixel image with the matched colors corresponding to the color gamut of the destination device.

`progressProc` A pointer to a callback function, supplied by the calling application or device driver, that monitors the color-matching progress or terminates the operation as your function matches the pixel map colors. Your `MyCMMatchPixMap` function must call this function at regular intervals to allow it to report progress to the user.

`refCon` A reference constant passed from the calling application or driver, which your `MyCMMatchPixMap` function must pass through as a parameter to calls it makes to the `MyCMBitmapCallBackProc` function.

DESCRIPTION

If your CMM supports this request code, your `MyCMMatchPixMap` function should be prepared to receive any of the pixel map types defined by QuickDraw.Your `MyCMMatchPixMap` function must match the colors of the pixel map image pointed to by `myPixMap` parameter to the destination profile's color gamut, replacing the original pixel colors with their corresponding colors as specified in the data color space of the destination device's color gamut.

Before the Component Manager calls your CMM with a ColorSync request to match the colors of a pixel map, it calls your CMM with a `kNCMMInit` or `kCMMConcatInit` request. Your CMM sets up the destination profile information during initialization in response to the `kNCMMInit` or `kCMMConcatInit` request code.

When the Component Manager calls your CMM with the `kCMMMatchPixMap` request code, it passes to your CMM in the `session` parameter a handle to your CMM's private storage for the calling applications's component instance. Your `MyCMMatchPixMap` function should use the profile data you set in your storage for this component instance to perform the color matching. If you used some other method to store profile data for this component instance when you initialized

the session, you should obtain the profile data you require for the color matching from that storage.

Your `MyCMMatchPixMap` function must call the progress function supplied by the calling application or device driver at regular intervals to allow it to report progress to the user on the color-matching session. Your `MyCMMatchPixMap` 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. The Apple-supplied CMM calls the progress function approximately every half second unless color matching occurs quickly enough to warrant not calling it at all; this happens when there is a small amount of data to be matched.

Here is the prototype for the `MyCMBitmapCallBackProc` function pointed to by the `progressProc` parameter:

```
pascal Boolean MyCMBitmapCallBackProc (long progress,
                                       void *refCon);
```

Each time your `MyCMMatchPixMap` function calls the `MyCMBitmapCallBackProc` function, it must pass to the function any data stored in the reference constant. When the Component Manager called your CMM with the `kCMMMatchPixMap` request code, it passed to your CMM the reference constant from the calling program.

Each time your function calls the `MyCMBitmapCallBackProc` function, your function must pass it a byte count in the `progress` parameter identifying the remaining number of bytes. The last time your `MyCMMatchPixMap` function calls the `MyCMBitmapCallBackProc` function, it must pass a byte count of 0 to indicate the completion of the matching process, signaling the progress function to perform any cleanup operations it requires.

The data color space of a pixel map is implicitly RGB. If the source and destination profiles' data color spaces (`dataColorSpace` field) are not also RGB, your function should not perform the color matching. Instead, your function should return an error.

If your CMM does not support a pixel map type that you receive, you can return an unimplemented error. In this case, the ColorSync Manager unpacks the colors of the pixel map and calls your `MyCMMatchColors` function, passing it the pixel map colors in a color list. You should avoid defaulting to this behavior, if possible, because it incurs overhead and slows down performance.

MyCMCheckPixMap

A CMM should respond to the `kCMMCheckPixMap` request code, but it is not required to do so. The ColorSync Manager sends this request code to your CMM on behalf of an application that called the `CWCheckPixMap` function. The ColorSync Manager dispatches this request to the Component Manager, which calls your CMM to service the request. A CMM typically responds to the `kCMMCheckPixMap` request code by calling a CMM-defined function (for example, `MyCMCheckPixMap`) to handle the request.

```
pascal CMError MyCMCheckPixMap(ComponentInstance CMSession,
                      const PixMap *myPixMap,
                      CMBitmapCallBackUPP progressProc,
                      BitMap *myBitMap, void *refCon);
```

`session` A handle to your CMM's storage for the instance of your component associated with the calling application or device driver.

`myPixMap` A pointer to a nonrelocatable pixel map whose colors are to be checked. A pixel map is a QuickDraw structure describing pixel data.

`progressProc` A pointer to a callback function, supplied by the calling application or device driver, that monitors the color-checking progress or terminates the operation as your function checks the pixel map colors. Your `MyCMCheckPixMap` function must call this function at regular intervals to allow it to report progress to the user.

`myBitMap` A QuickDraw bitmap whose boundaries equal those of the pixel map indicated by the `myPixMap` parameter. Your `MyCMCheckPixMap` function must set a pixel to 1 if the corresponding pixel of the pixel map indicated by `myPixMap` is out of gamut.

`refCon` A reference constant passed from the calling application or driver, which your `MyCMCheckPixMap` function must pass through as a parameter to calls it makes to the `MyCMBitmapCallBackProc` function.

DESCRIPTION

If your CMM supports this request code, your `MyCMCheckPixMap` function should be prepared to receive any of the pixel map types defined by QuickDraw.Your `MyCMCheckPixMap` function must check the colors of the pixel map image pointed to by the `myPixMap` parameter against the color gamut of the destination profile to determine if the colors are within the gamut. If a pixel color of the pixel map indicated by `myPixMap` is out of gamut, your function must set to 1 the corresponding pixel of the bitmap indicated by `myBitMap`. The ColorSync Manager returns the bitmap showing the gamut check results to the calling application or device driver.

Before the Component Manager calls your CMM with a ColorSync request to check the colors of a pixel map, it calls your CMM with a `kNCMMInit` or `kCMMConcatInit` request. Your CMM sets up the destination profile information during initialization in response to the `kNCMMInit` or `kCMMConcatInit` request code.

When the Component Manager calls your CMM with the `kCMMCheckPixMap` request code, it passes to your CMM in the `session` parameter a handle to your CMM's private storage for the calling applications's component instance. Your `MyCMCheckPixMap` function should use the profile data you set in your storage for this component instance. If you used some other method to store profile data for this component instance when you initialized the session, you should obtain the profile data you require for the color-checking process from that storage.

Your `MyCMMatchPixMap` function must call the progress function supplied by the calling application or device driver at regular intervals to allow it to report progress to the user on the color-checking session. Your `MyCMCheckPixMap` 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 color-checking process. The Apple-supplied CMM calls the progress function approximately every half second unless color checking occurs quickly enough to warrant not calling it at all; this happens when there is a small amount of data to be matched.

Here is the prototype for the `MyCMBitmapCallBackProc` function pointed to by the `progressProc` parameter:

```
pascal Boolean MyCMBitmapCallBackProc (long progress,
                                       void *refCon);
```

Each time your `MyCMCheckPixMap` function calls the `MyCMBitmapCallBackProc` function, it must pass to the function any data stored in the reference constant. When the Component Manager called your CMM with the `kCMMCheckPixMap` request code, it passed to your CMM the reference constant from the calling program.

Each time your function calls the `MyCMBitmapCallBackProc` function, your function must pass it a byte count in the `progress` parameter identifying the remaining number of bytes to be checked. As your `MyCMCheckPixMap` function checks the pixels of the `myPixMap` map, it should set the corresponding pixel of `myBitMap` to 0 if the color is in gamut and 1 if it is out of gamut. The last time your `MyCMCheckPixMap` function calls the `MyCMBitmapCallBackProc` function, it must pass a byte count of 0 to indicate the completion of the color-checking process, signaling the progress function to perform any cleanup operations it requires.

The data color space of a pixel map is implicitly RGB. If the source and destination profiles' data color spaces (`dataColorSpace` field) are not also RGB, your function should not perform the color check. Instead, your function should return an error.

If your CMM does not support a pixel map type that you receive, you can return an unimplemented error. In this case, the ColorSync Manager unpacks the colors of the pixel map and calls your `MyCMMatchColors` function, passing it the pixel map colors in a color list. You should avoid defaulting to this behavior, if possible, because it incurs overhead and slows down performance.

MyCMNewLinkProfile

A CMM should respond to the `kCMMNewLinkProfile` request code, but it is not required to do so. The ColorSync Manager sends this request code to your CMM on behalf of an application that called the `CWNewLinkProfile` function. The ColorSync Manager dispatches this request to the Component Manager, which calls your CMM to service the request. A CMM typically responds to the

`kCMMNewLinkProfile` request code by calling a CMM-defined function (for example, `MyCMNewLinkProfile`) to handle the request.

```
pascal CMError MyCMNewLinkProfile(ComponentInstance CMSession,
                        CMProfileRef *prof, const
                        CMProfileLocation *targetLocation,
                        CMConcatProfileSet *profileSet);
```

`session` A handle to your CMM's storage for the instance of your component associated with the calling application or device driver.

`prof` A reference to a device-linked profile of type `DeviceLink`. Your `MyCMNewLinkProfile` function creates this profile, opens it to obtain a reference to it, and returns the profile reference in this parameter. The profile may be a file-based profile or a handle-based profile. It must not be a pointer-based profile or a temporary profile.

`targetLocation`
The location specification for the resulting profile, which your function returns. This is the file specification where you created the profile. For information on how to specify the location, see the sections "Profile Location Union" and "Profile Location Structure", both in the chapter "ColorSync Manager Reference for Applications and Device Drivers" in the *Advanced Color Imaging Reference* on the enclosed CD.

`profileSet` A pointer to a data structure of type `CMConcatProfileSet` containing an array of profiles. Your function must include these profiles in order in any device-linked profile it creates. The profiles in the array are in processing order—source through destination. The `profileSet` field of the data structure contains the array. For a description of the `CMConcatProfileSet` data structure, see "Concatenated Profile Set Structure" in the chapter "ColorSync Manager Reference for Applications and Device Drivers" in the *Advanced Color Imaging Reference* on the enclosed CD.

DESCRIPTION

Your `MyCMNewLinkProfile` function 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. For information about profiles of type `DeviceLink`, see the chapter "ColorSync Manager Reference for Applications and Device Drivers" in the *Advanced Color Imaging Reference* on the enclosed CD. You must adhere to the requirements for device-linked profiles described in same chapter.

After your function creates the device-linked profile, it must open the profile and return a reference to the profile in the `prof` parameter.

The *International Color Consortium Profile Format Specification,* version 2.0, document revision 3.x, also describes device-linked profiles. 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.

MyCMMGetPS2ColorSpace

A CMM may respond to the `kCMMGetPS2ColorSpace` request code, but it is not required to do so. The ColorSync Manager sends this request code to your CMM on behalf of an application that called the `CMGetPS2ColorSpace` function. The ColorSync Manager dispatches this request to the Component Manager, which calls your CMM to service the request. A CMM typically responds to the `kCMMGetPS2ColorSpace` request code by calling a CMM-defined function (for example, `MyCMMGetPS2ColorSpace`) to handle the request.

```
pascal CMError MyCMMGetPS2ColorSpace(ComponentInstance CMSession,
                    CMProfileRef srcProf,
                    unsigned long flags,
                    CMFlattenUPP proc,
                    void *refCon);
```

`session` A handle to your CMM's storage for the instance of your component associated with the calling application or device driver.

`srcProf` A profile reference to the source profile from which you must obtain or derive the color space element data.

`flags` Reserved for future use.

`proc` A pointer to a `MyColorSyncDataTransfer` function supplied by the calling application or device driver. Your `MyCMMGetPS2ColorSpace` function calls this function repeatedly as necessary until you have passed all the source profile's color space element data to this function.

`refCon` A reference constant, containing data specified by the calling application or device driver, that your `MyCMMGetPS2ColorSpace` function must pass to the `MyColorSyncDataTransfer` function.

DESCRIPTION

Only for special cases should a custom CMM need to support this request code. If your CMM supports this function, your `MyCMMGetPS2ColorSpace` function must obtain or derive the color space element data from the source profile whose reference is passed to your function in the `srcProf` parameter.

The color space data may be assigned to the PostScript Level 2 color space array (`ps2CSATag`) tag in the source profile. The byte stream containing the color space element data that your function passes to the `MyColorSyncDataTransfer` function is used as the operand to the PostScript `setColorSpace` operator.

Your function must allocate a `data` buffer in which to pass the color space element data to the `MyColorSyncDataTransfer` function supplied by the calling application or driver. Your `MyCMMGetPS2ColorSpace` function must call the `MyColorSyncDataTransfer` function repeatedly until you have passed all the data to it. Here is the prototype for the `MyColorSyncDataTransfer` function pointed to by the `proc` parameter:

```
pascal OSErr MyColorSyncDataTransfer(long command, long *size,
                                     void *data, void *refCon);
```

Your `MyCMMGetPS2ColorSpace` function communicates with the `MyColorSyncDataTransfer` function using a command parameter to identify the operation to be performed. Your function should call the `MyColorSyncDataTransfer` function first with the `openWriteSpool` command to direct the `MyColorSyncDataTransfer` function to the begin the process of writing the profile color space element data you pass it in the `data` buffer. Next, you should call the `MyColorSyncDataTransfer` function with the `writeSpool` command. After the `MyColorSyncDataTransfer` function returns in the `size` parameter the amount of data it actually wrote, you should call the

`MyColorSyncDataTransfer` function again with the `writeSpool` command, repeating this process as often as necessary until all the color space data is transferred. After the data is transferred, you should call the `MyColorSyncDataTransfer` function with the `closeSpool` command.

When your function calls the `MyColorSyncDataTransfer` function, it passes in the `data` buffer the profile data to be transferred to the `MyColorSyncDataTransfer` function and the size in bytes of the buffered data in the `size` parameter. The `MyColorSyncDataTransfer` function may not always write all the data you pass it in the `data` buffer. Therefore, on return the `MyColorSyncDataTransfer` function command passes back in the `size` parameter the number of bytes it actually wrote. Your `MyCMMGetPS2ColorSpace` function keeps track of the number of bytes of remaining color space element data.

Each time your `MyCMMGetPS2ColorSpace` function calls the `MyColorSyncDataTransfer` function, you pass it the reference constant passed to your function in the reference constant parameter.

SEE ALSO

For information about PostScript operations, see the *PostScript Language Manual,* second edition.

MyCMMGetPS2ColorRenderingIntent

A CMM may respond to the `kCMMGetPS2ColorRenderingIntent` request code, but it is not required to do so. The ColorSync Manager sends this request code to your CMM on behalf of an application that called the `CMGetPS2ColorRenderingIntent` function. The ColorSync Manager dispatches this request to the Component Manager, which calls your CMM to service the request. A CMM typically responds to the `kCMMGetPS2ColorRenderingIntent` request code by calling a CMM-defined function (for example, `MyCMMGetPS2ColorRenderingIntent`) to handle the request.

```
pascal CMError MyCMMGetPS2ColorRenderingIntent(ComponentInstance
                    CMSession, CMProfileRef srcProf,
                    unsigned long flags,
                    CMFlattenUPP proc, void *refCon);
```

`CMSession` A handle to your CMM's storage for the instance of your component associated with the calling application or device driver.

`srcProf` A profile reference to the source profile whose header contains the rendering intent.

`flags` Reserved for future use.

`proc` A pointer to a function supplied by the calling application or device driver. Your `MyCMMGetPS2ColorRenderingIntent` function calls this function repeatedly as necessary until you have passed all the source profile's rendering intent data to this function.

`refCon` A reference constant, containing data specified by the calling application or device driver, that your `MyCMMGetPS2ColorRenderingIntent` function must pass to the `MyColorSyncDataTransfer` function.

DESCRIPTION

Only for special cases should a custom CMM need to support this request code. If your CMM supports this function, your `MyCMMGetPS2ColorRenderingIntent` function must obtain the rendering intent from the source profile whose reference is passed to your function in the `srcProf` parameter. The byte stream containing the rendering intent data that your function passes to the `MyColorSyncDataTransfer` function is used as the operand to the PostScript `findRenderingIntent` operator.

Your function must allocate a `data` buffer in which to pass the rendering intent data to the `MyColorSyncDataTransfer` function supplied by the calling application or driver. Your `MyCMMGetPS2ColorRenderingIntent` function must call the `MyColorSyncDataTransfer` function repeatedly until you have passed all the data to it.

Here is the prototype for the `MyColorSyncDataTransfer` function pointed to by the `proc` parameter:

```
pascal OSErr MyColorSyncDataTransfer(long command, long *size,
                                     void *data, void *refCon);
```

Your `MyCMMGetPS2ColorRenderingIntent` function communicates with the `MyColorSyncDataTransfer` function using a command parameter to identify the

operation to be performed. Your function should call the `MyColorSyncDataTransfer` function first with the `openWriteSpool` command to direct the `MyColorSyncDataTransfer` function to the begin the process of writing the profile color-rendering intent element data you pass it in the `data` buffer. Next, you should call the `MyColorSyncDataTransfer` function with the `writeSpool` command. After the `MyColorSyncDataTransfer` function returns in the `size` parameter the amount of data it actually read, you should call the `MyColorSyncDataTransfer` function again with the `writeSpool` command, repeating this process as often as necessary until all the color-rendering intent data is transferred. After the data is transferred, you should call the `MyColorSyncDataTransfer` function with the `closeSpool` command.

When your function calls the `MyColorSyncDataTransfer` function, it passes in the `data` buffer the profile data to be transferred to the `MyColorSyncDataTransfer` function and the size in bytes of the buffered data in the `size` parameter. The `MyColorSyncDataTransfer` function may not always write all the data you pass it in the `data` buffer. Therefore, on return the `MyColorSyncDataTransfer` function command passes back in the `size` parameter the number of bytes it actually wrote. Your `MyCMMGetPS2ColorRenderingIntent` function keeps track of the number of bytes of remaining color-rendering intent element data.

Each time your `MyCMMGetPS2ColorRenderingIntent` function calls the `MyColorSyncDataTransfer` function, you pass it the reference constant passed to your function in the reference constant parameter.

SEE ALSO

For information about PostScript operations, see the *PostScript Language Manual,* second edition.

MyCMMGetPS2ColorRendering

A CMM may respond to the `kCMMGetPS2ColorRendering` request code, but it is not required to do so. The ColorSync Manager sends this request code to your CMM on behalf of an application that called the `CMGetPS2ColorRendering` function. The ColorSync Manager dispatches this request to the Component Manager, which calls your CMM to service the request. A CMM typically responds to the `kCMMGetPS2ColorRendering` request code by calling a

CMM-defined function (for example, `MyCMMGetPS2ColorRendering`) to handle the request.

```
pascal CMError MyCMMGetPS2ColorRendering(ComponentInstance CMSession,
                     CMProfileRef srcProf, CMProfileRef dstProf,
                     unsigned long flags,CMFlattenUPP proc,
                     void *refCon);
```

`CMSession` A handle to your CMM's private storage for the instance of your component associated with the calling application or device driver.

`srcProf` A profile reference to the source profile whose header indicates the rendering intent for generating the CRD.

`dstProf` A profile reference to the destination profile from which you obtain or derive the color-rendering dictionary (CRD).

`flags` Reserved for future use.

`proc` A pointer to a function supplied by the calling application or device driver. Your `MyCMMGetPS2ColorRendering` function calls this function repeatedly as necessary until you have passed all the color-rendering dictionary (CRD) element data to this function.

`refCon` A reference constant, containing data specified by the calling application or device driver, that your `MyCMMGetPS2ColorRendering` function must pass to the `MyColorSyncDataTransfer` function.

DESCRIPTION

Only for special cases should a custom CMM need to support this request code. If your CMM supports this function, your `MyCMMGetPS2ColorRendering` function must obtain the rendering intent from the header of the source profile identified by the `srcProf` parameter. The rendering intent identifies the CRD data that you must obtain or derive from the destination profile whose reference is passed to your function in the `dstProf` parameter. The byte stream containing the specified rendering intent's CRD data that your function passes to the `MyColorSyncDataTransfer` function is used as the operand to the PostScript `setColorRendering` operator.

A profile may contain tags that specify the CRD data for each rendering intent. A profile's `ps2CRD0Tag` element data contains the CRD for perceptual rendering. A profile's `ps2CRD1Tag` contains the CRD for relative colorimetric rendering. A profile's `ps2CS2Tag` contains the CRD for saturation rendering. A profile's `ps2CS3Tag` contains the CRD for absolute colorimetric rendering. If the profile does not contain a CRD tag, your CMM should create the CRD from the destination profile using the rendering intent specified by the source profile.

Your function must allocate a `data` buffer in which to pass the CRD data to the `MyColorSyncDataTransfer` function supplied by the calling application or driver. Your `MyCMMGetPS2ColorRendering` function must call the `MyColorSyncDataTransfer` function repeatedly until you have passed all the data to it. Here is the prototype for the `MyColorSyncDataTransfer` function pointed to by the `proc` parameter:

```
pascal OSErr MyColorSyncDataTransfer(long command, long *size,
                                     void *data, void *refCon);
```

Your `MyCMMGetPS2ColorRendering` function communicates with the `MyColorSyncDataTransfer` function using a command parameter to identify the operation to be performed. Your function should call the `MyColorSyncDataTransfer` function first with the `openWriteSpool` command to direct the `MyColorSyncDataTransfer` function to the begin the process of writing the profile CRD data you pass it in the `data` buffer. Next, you should call the `MyColorSyncDataTransfer` function with the `writeSpool` command. After the `MyColorSyncDataTransfer` function returns in the `size` parameter the amount of data it actually wrote, you should call the `MyColorSyncDataTransfer` function again with the `writeSpool` command, repeating this process as often as necessary until all the CRD data is transferred. After the data is transferred, you should call the `MyColorSyncDataTransfer` function with the `closeSpool` command.

When your function calls the `MyColorSyncDataTransfer` function, it passes in the `data` buffer the profile data to be transferred to the `MyColorSyncDataTransfer` function and the size in bytes of the buffered data in the `size` parameter. The `MyColorSyncDataTransfer` function may not always write all the data you pass it in the `data` buffer. Therefore, on return the `MyColorSyncDataTransfer` function command passes back in the `size` parameter the number of bytes it actually wrote. Your `MyCMMGetPS2ColorRendering` function keeps track of the number of bytes of remaining CRD data.

Each time your `MyCMMGetPS2ColorRendering` function calls the `MyColorSyncDataTransfer` function, you pass it the reference constant passed to your function in the reference constant parameter.

SEE ALSO

For information about PostScript operations, see the *PostScript Language Manual,* second edition.

MyCMMGetPS2ColorRenderingVMSize

A CMM may respond to the `kCMMGetPS2ColorRenderingVMSize` request code, but it is not required to do so. The ColorSync Manager sends this request code to your CMM on behalf of an application that called the `CMGetPS2ColorRenderingVMSize` function. The ColorSync Manager dispatches this request to the Component Manager, which calls your CMM to service the request. A CMM typically responds to the `kCMMGetPS2ColorRenderingVMSize` request code by calling a CMM-defined function (for example, `MyCMMGetPS2ColorRenderingVMSize`) to handle the request.

```
pascal CMError MyCMMGetPS2ColorRenderingVMSize(ComponentInstance
                         CMSession, CMProfileRef srcProf,
                         CMProfileRef dstProf,
                         unsigned long vmSize);
```

`CMSession` A handle to your CMM's storage for the instance of your component associated with the calling application or device driver.

`srcProf` A profile reference to the source profile specifying the rendering intent to be used.

`dstProf` A profile reference to the destination printer profile from which you obtain or assess the virtual memory (VM) size of the CRD.

`vmSize` The VM size of the CRD, returned by the function.

DESCRIPTION

Only for special cases should a custom CMM need to support this request code. If your CMM supports this function, your `MyCMMGetPS2ColorRenderingVMSize` function must obtain the maximum VM size of the CRD for the rendering intent specified by the source profile.

Your function must return the VM size in the `vmSize` parameter. (In turn, the ColorSync Manager returns the VM size to the calling application or device driver.) The CRD whose maximum size you return must be that of the dictionary for the rendering intent specified by the source profile.

If the destination profile contains the Apple-defined private tag `'psvm'`, described later in this section, 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. In this case, the assessment may be larger than the actual maximum VM size.

The `CMPS2CRDVMSizeType` data type defines the Apple-defined `'psvm'` optional tag that a 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.

The `CMIntentCRDVMSize` data type defines the rendering intent and its maximum VM size:

```
struct CMIntentCRDVMSize {
    long            rendering   Intent;
    unsigned long   VMSize;
};
```

For example, a rendering intent might be 0 and its VM size 120 KB.

Constant descriptions

`renderingIntent` The rendering intent whose CRD VM size you want to obtain. Rendering intent values are

`0` (`cmPerceptual`)

`1` (`cmRelativeColorimetric`)

`2` (`cmSaturation`)

`3` (`cmAbsoluteColorimetric`)

`VMSize` The VM size of the CRD for the rendering intent specified for the `renderingIntent` field.

The `CMPS2CRDVMSizeType` data type for the tag includes an array containing one or more members of type `CMIntentCRDVMSize`:

```
struct CMPS2CRDVMSizeType {
    OSType  typeDescriptor;
    unsigned long   reserved;
    unsigned long   count;
    CMIntentCRDVMSize   intentCRD[1];
};
```

Constant descriptions

`typeDescriptor` The `'psvm'` tag signature.

`reserved` Reserved for future use.

`count` The number of entries in the `intentCRD` array.

`CMIntentCRDVMSize` A variable-sized array of four or more members defined by the `CMIntentCRDSize` data type.

MyCMMFlattenProfile

A CMM may respond to the `kCMMFlattenProfile` request code, but it is not required to do so. For most CMMs, the Apple-default CMM can handle this request code. The ColorSync Manager sends this request code to your CMM on behalf of an application or device driver that called the `CMFlattenProfile` function. The ColorSync Manager dispatches this request to the Component Manager which calls your CMM to service the request. A CMM that handles the `kCMMFlattenProfile` request code typically responds by calling a CMM-defined function (for example, `MyCMMFlattenProfile`).

```
pascal CMError MyCMMFlattenProfile (ComponentInstance CMSession,
                    CMProfileRef prof, unsigned
                    long flags, CMFlattenUPP
                    proc, void *refCon);
```

`CMSession` A handle to your CMM's storage for the instance of your component associated with the calling application or device driver.

`prof` A reference to the profile to be flattened.

`flags` Reserved for future use.

`proc` A pointer to the `MyColorSyncDataTransfer` function supplied by the calling application or device driver to perform the low-level data transfer. Your `MyCMMFlattenProfile` function calls this function repeatedly as necessary until all the profile data is transferred.

`refCon` A reference constant containing data specified by the calling application or device driver.

DESCRIPTION

Only in rare circumstances should a custom CMM need to support this request code. The process of flattening a profile is complex, and the Apple-supplied default CMM handles this process adequately for most cases. A custom CMM might respond to this request code if the CMM provides special services such as profile data encryption or compression, for example. Read the rest of this description if your CMM handles this request code.

Your `MyCMMFlattenProfile` function must extract the profile data from the profile to be flattened, identified by the `prof` parameter, and pass the profile data to the function specified in the `proc` parameter.

Your `MyCMMFlattenProfile` function calls the `MyColorSyncDataTransfer` function supplied by the calling application. Here is the prototype for the `MyColorSyncDataTransfer` function pointed to by the `proc` parameter:

```
pascal OSErr MyColorSyncDataTransfer(long command, long *size,
                                     void *data, void *refCon);
```

Your `MyCMMFlattenProfile` function communicates with the `MyColorSyncDataTransfer` function using a command parameter to identify the operation to be performed. Your function should call the `MyColorSyncDataTransfer` function first with the `openWriteSpool` command to direct the `MyColorSyncDataTransfer` function to the begin the process of writing the profile data you pass it in the `data` buffer. Next, you should call the `MyColorSyncDataTransfer` function with the `writeSpool` command. After the

MyColorSyncDataTransfer function returns in the size parameter the amount of data it actually wrote, you should call the MyColorSyncDataTransfer function again with the writeSpool command, repeating this process as often as necessary until all the profile data is transferred. After the data is transferred, you should call the MyColorSyncDataTransfer function with the closeSpool command.

When your function calls the MyColorSyncDataTransfer function, it passes in the data buffer the profile data to be transferred to the MyColorSyncDataTransfer function and the size in bytes of the buffered data in the size parameter. The MyColorSyncDataTransfer function may not always write all the data you pass it in the data buffer. Therefore, on return the MyColorSyncDataTransfer function command passes back in the size parameter the number of bytes it actually wrote. Your function keeps track of the number of bytes of remaining profile data.

Your MyCMMFlattenProfile function is responsible for obtaining the profile data from the profile, allocating a buffer in which to pass the data to the MyColorSyncDataTransfer function, and keeping track of the amount of remaining data to be transferred to the MyColorSyncDataTransfer function.

Each time your MyCMMFlattenProfile function calls the MyColorSyncDataTransfer function, you pass it the reference constant passed to your function in the reference constant parameter.

MyCMMUnflattenProfile

A CMM may respond to the kCMMUnflattenProfile request code, but it is not required to do so. For most CMMs, the Apple-default CMM can handle this request code. The ColorSync Manager sends this request code to your CMM on behalf of an application or device driver that called the CMUnflattenProfile function. The ColorSync Manager dispatches this request to the Component Manager, which calls your CMM to service the request. A CMM that handles the kCMMUnflattenProfile request code typically responds by calling a CMM-defined function (for example, MyCMMUnflattenProfile).

```
pascal CMError MyCMMUnflattenProfile (ComponentInstance CMSession,
                        FSSpec *resultFileSpec,
                        CMFlattenUPP proc,
                        void *refCon);
```

`CMSession` A handle to your CMM's storage for the instance of your component associated with the calling application or device driver.

`resultFileSpec`
A file specification for the profile file. This is a temporary file specification. You must create this temporary file, which is returned to the calling application or device driver. The calling application or driver is responsible for disposing of the file when finished with it.

`proc` A pointer to a function supplied by the calling application or device driver to perform the low-level data transfer. Your `MyCMMFlattenProfile` function calls this function repeatedly as necessary until all the profile data is transferred.

`refCon` A reference constant containing data specified by the calling application program.

DESCRIPTION

Only in rare circumstances should a custom CMM need to support this request code. The process of unflattening a profile is complex, and the Apple-supplied default CMM handles this process adequately for most cases. A custom CMM might respond to this request code if the CMM provides special services such as profile data encryption or compression, for example. Read the rest of this description if your CMM handles this request code.

Your `MyCMMUnflattenProfile` function must create a file with a unique name in which to store the profile data. (You should create this file in the temporary items folder.) The ColorSync Manager returns the temporary file specification to the calling application or device driver.

To obtain the profile data, your `MyCMMUnflattenProfile` function calls the `MyColorSyncDataTransfer` function supplied by the calling application or device driver. Here is the prototype for the `MyColorSyncDataTransfer` function pointed to by the `proc` parameter:

```
pascal OSErr MyColorSyncDataTransfer (long command, long *size,
                                       void *data, void *refCon);
```

Before calling the `MyColorSyncDataTransfer` function, your `MyCMMUnflattenProfile` function must allocate a buffer to hold the profile data

returned to you from the MyColorSyncDataTransfer function in the data parameter.

Your MyCMMUnflattenProfile function communicates with the MyColorSyncDataTransfer function using a command parameter to identify the operation to be performed. Your function should call the MyColorSyncDataTransfer function first with the openReadSpool command to direct the MyColorSyncDataTransfer function to begin the process of transferring data. Following this, you should call the MyColorSyncDataTransfer function with the readSpool command as often as necessary until the MyColorSyncDataTransfer function has passed your function all the profile data from the graphics file. After you have received all the profile data, your function should call the MyColorSyncDataTransfer function with the closeSpool command.

Each time you call the MyColorSyncDataTransfer function, you should pass it a pointer to the data buffer you created, the size in bytes of the profile data to be returned to you in the buffer, and the reference constant passed to you from the calling application.

On return, the MyColorSyncDataTransfer function passes to you the profile data that your function must write to the temporary file that you created for the new profile file. The MyColorSyncDataTransfer function will not always transfer the number of bytes of profile data you requested. Therefore, the MyColorSyncDataTransfer function returns in the size parameter the number of bytes of profile data it actually returned in the data buffer.

The profile file you create is returned to the calling application or device driver in the resultFileSpec parameter. Your MyCMUnflattenProfile function must identify the profile size and maintain a counter tracking the amount of data transferred to you and the amount of remaining data in order to determine when to call the MyColorSyncDataTransfer function with the closeSpool command. To determine the profile size, your function can obtain the profile header, which specifies the size.

The calling application or device driver uses the reference constant to pass to the MyColorSyncDataTransfer function information the MyColorSyncDataTransfer function requires to transfer the data.

CHAPTER 5

Color Manager Reference

Contents

Constants and Data Types 5-3
- ITab 5-3
- SProcRec 5-4
- CProcRec 5-4
- ReqListRec 5-5

Color Manager Functions 5-5
- Managing Colors 5-5
  - Color2Index 5-5
  - Index2Color 5-6
  - InvertColor 5-7
  - RealColor 5-7
  - GetSubTable 5-8
  - MakeITable 5-9
- Managing Color Tables 5-10
  - GetCTSeed 5-10
  - ProtectEntry 5-10
  - ReserveEntry 5-11
  - SetEntries 5-12
  - SaveEntries 5-13
  - RestoreEntries 5-14
- Operations on Search and Complement Functions 5-15
  - AddSearch 5-16
  - AddComp 5-16
  - DelSearch 5-16
  - DelComp 5-17
  - SetClientID 5-17
- Application-Defined Functions 5-17

This document was created with FrameMaker 4.0.4

`MySearchProc` **5-17**
`MyCompProc` **5-18**

This section presents a complete reference to the data types and functions of the Color Manager.

Constants and Data Types

The Color Manager contains data structures for holding inverse table information and links in the chains of custom search and complement functions.

ITab

The `ITab` data structure contains the inverse table information that the Color Manager uses for fast mapping of RGB color values.

```
struct ITab {
    long            iTabSeed;       /* copy of color table seed */
    short           iTabRes;        /* resolution of table */
    unsigned char   iTTable[1];     /* byte color table index values */
};
typedef  struct ITab ITab;
typedef ITab *ITabPtr, **ITabHandle;
```

Field descriptions

`iTabSeed` — The `iTabSeed` value, initially set from the corresponding CLUT's `ctSeed` field. If at any time these don't match, then the color table was changed, and the inverse table needs to be rebuilt.

`iTabRes` — The resolution of this inverse table.

`iTabTable` — An array of index values. The size of the `iTabTable` field in bytes is $2^{3*iTabRes}$.

This document was created with FrameMaker 4.0.4

SProcRec

The SProcRec data structure contains a pointer to a custom search function and a handle to the next SProcRec data structure in the function list.

```
struct SProcRec {
    Handle              nxtSrch;    /* handle to next SProcRec */
    ColorSearchProcPtr  srchProc;   /* pointer to search function */
};
typedef struct SProcRec SProcRec;
typedef SProcRec *SProcPtr, **SProcHndl;
```

Field descriptions

nxtSrch — A handle to the next SProcRec data structure in the chain of search functions.

srchProc — A pointer to a custom search function (described on page 5-17).

CProcRec

The CProcRec data structure contains a pointer to a custom complement function and a pointer to the next complement function in the list.

```
struct CProcRec {
    Handle          nxtComp;    /* handle to next CProcRec */
    ColorComplementProcPtr
                    compProc;   /* pointer to complement function */
};
typedef struct CProcRec CProcRec;
typedef CProcRec *CProcPtr, **CProcHndl;
```

Field descriptions

nxtComp — A handle to the next CPRocRec data structure in the list.

compProc — A pointer to a complement function, as described on page 5-18.

ReqListRec

The ReqListRec data structure is a parameter to the SaveEntries function by which you can describe color table entries to be saved.

```
struct ReqListRec {
    short reqLSize;     /* request list size minus 1 */
    short reqLData[1]   /* request list data */
};
typedef struct ReqListRec ReqListRec;
```

Field descriptions

reqLSize — The size of this ReqListRec data structure minus one.

reqLData — An array of integers representing offsets into a color table.

Color Manager Functions

The Color Manager provides functions for color management, color table management, and inverse table management. System software, such as Color QuickDraw and the Palette Manager, calls these functions automatically; applications should generally never need to call these functions, which are described here for completeness.

Managing Colors

You can find the index to the best approximation of a single color with the Color2Index function, and you can find the indexes to the best approximation of a set of colors with the GetSubTable function.

Color2Index

System software uses the Color2Index function to obtain the index of the best available approximation for a given color in the color table of the current GDevice data structure.

```
pascal long Color2Index (const RGBColor *myColor);
```

`myColor` A pointer to the RGB color value to be approximated.

DESCRIPTION

The `Color2Index` function returns the index of the best approximation for a given color that is available in the color table of the current `GDevice` data structure. Note that `Color2Index` returns a long integer, in which the low-order word is the index value; the high-order word contains zeros.

You should not call `Color2Index` from within a custom search function (described on page 5-17).

Index2Color

System software uses the `Index2Color` function to obtain the `RGBColor` data structure corresponding to an index value in the color table of the current `GDevice` data structure.

```
pascal void Index2Color (
                    long index,
                    RGBColor *aColor);
```

`index` The index value whose color entry is sought.

`aColor` A pointer to the returned `RGBColor` data structure.

DESCRIPTION

The `Index2Color` function returns the `RGBColor` data structure corresponding to an index value in the color table of the current `GDevice` data structure. For the index value you should supply a long integer in which the high-order word is padded with zeros.

InvertColor

System software uses the `InvertColor` function to find the complement of an `RGBColor` data structure.

```
pascal void InvertColor (RBGColor *myColor);
```

`myColor` The `RGBColor` data structure for which the complement is to be found.

DESCRIPTION

The `InvertColor` function returns the complement of an absolute color, using the list of complement functions in the current device data structure. The default complement function uses the one's complement of each component of the requested color.

RealColor

System software uses the `RealColor` function to determine whether a given `RGBColor` data structure actually exists in the current device's color table.

```
pascal Boolean RealColor (const RGBColor *color);
```

`color` The `RGBColor` data structure to be tested.

DESCRIPTION

The `RealColor` function determines whether the color is available in the current `GDevice` data structure's CLUT, basing its search on the current resolution of the inverse table. For example, if the current value of the `iTabRes` field is 4, `RealColor` returns `true` if there exists a color that exactly matches the top 4 bits of red, green, and blue. (The `iTabRes` field of the inverse table is described on page 5-3.)

GetSubTable

System software uses the `GetSubTable` function to search one color table for the best matches to colors in another color table. You can use this function to determine the best indexes in the current `GDevice` data structure's CLUT for a set of colors in your application's color table.

```
pascal void GetSubTable (
                    CTabHandle myColors,
                    short iTabRes,
                    CTabHandle targetTbl);
```

`myColors` A handle to a color table containing the colors for which you want matches.

`iTabRes` The resolution of the inverse table to be used.

`targetTbl` A handle to a color table whose colors are to be matched.

DESCRIPTION

The `GetSubTable` function searches one color table for the best matches to colors in another color table. Supply the colors you want matched in the `myColors` parameter. Supply the color table to be searched in the `targetTbl` parameter. `GetSubTable` stores indexes from the color table in `targetTbl` in the `value` field of the color table in the `myColors` parameter.

The Color Manager uses the `Color2Index` function for each `RGBColor` data structure in the color table of the `myColors` parameter. It determines the best match in the target table and stores that index value in the color table of the `myColors` parameter.

If you supply `nil` for `targetTbl`, then the Color Manager searches the current `GDevice` data structure's CLUT, and uses its inverse table. Otherwise a temporary inverse table is built, with a resolution of the value in the `iTabRes` parameter.

SPECIAL CONSIDERATIONS

Depending on the requested resolution, building the inverse table can require large amounts of temporary space in the application heap: twice the size of the table itself, plus a fixed overhead of 3–15 KB for each inverse table resolution.

MakeITable

System software uses the `MakeITable` function to generate an inverse table for a color table.

```
pascal void MakeITable (
                    CTabHandle cTabH,
                    ITabHandle iTabH,
                    short res);
```

`cTabH` The color table for which an inverse table is to be generated.

`iTabH` The generated inverse table.

`res` The resolution needed for the inverse table.

DESCRIPTION

The `MakeITable` function generates an inverse table based on the current contents of the color table pointed to by the `colorTab` parameter, with a resolution specified by the value in the `res` parameter. Reserved color table pixel values are not included in the resulting color table. `MakeITable` tests its input parameters and returns an error in `QDError` if the resolution is less than three or greater than five. Passing `nil` in the `colorTab` or `inverseTab` parameter substitutes an appropriate handle from the current `GDevice` data structure, while passing 0 in the `res` parameter substitutes the current `GDevice` data structure's inverse table resolution. These defaults can be used in any combination with explicit values, or with `nil` parameter values.

This function allows maximum precision in mapping colors, even if colors in the color table differ by less than the resolution of the inverse table. Five-bit inverse tables are not needed when drawing in normal Color QuickDraw modes. However, Color QuickDraw transfer modes such as add, subtract, and blend may require a 5-bit inverse table for best results with certain color tables. `MakeITable` returns an error in `QDError` if the destination inverse table memory cannot be allocated. The `'mitq'` resource governs how much memory is allocated for temporary internal structures; this resource type is for internal use only.

SPECIAL CONSIDERATIONS

Depending on the requested resolution, building the inverse table can require large amounts of temporary space in the application heap: twice the size of the table itself, plus a fixed overhead of 3–15 KB for each inverse table resolution.

Managing Color Tables

The functions in this section enable a specialized application to obtain a seed value for a color table you create so that the Color Manager can note when the table is changed, and to change the values and protection of color table entries.

GetCTSeed

You can use the `GetCTSeed` function to obtain a unique seed value for a color table created by your application.

```
pascal long GetCTSeed (void);
```

DESCRIPTION

The `GetCTSeed` function returns a unique seed value that you can use in the `ctSeed` field of a color table created by your application. The seed value guarantees that the color table is recognized as distinct from the destination, and that color table translation is performed properly. The return value is greater than the value stored in the constant `minSeed`.

ProtectEntry

You can use the `ProtectEntry` function to add protection to or remove protection from an entry in the current `GDevice` data structure's color table.

```
pascal void ProtectEntry (
                          short index,
                          Boolean protect);
```

`index` The index to the entry whose protection is to be changed.

`protect` A Boolean value: specify `true` to protect the entry, `false` to remove protection.

DESCRIPTION

The `ProtectEntry` function adds or removes protection for an entry in the current `GDevice` data structure's color table, depending on the value of the `protect` parameter. A protected entry can't be changed by other applications. `ProtectEntry` returns a protection error in `QDErr` if you attempt to protect an already protected entry. However, it can remove protection from any entry.

ReserveEntry

You can use the `ReserveEntry` function to reserve or remove reservation from an entry in the current `GDevice` data structure's color table.

```
pascal void ReserveEntry (
                    short index,
                    Boolean reserve);
```

`index` The index to the entry.

`reserve` A Boolean value, `true` to reserve the entry, `false` to remove the reservation.

DESCRIPTION

The `ReserveEntry` function reserves or removes the reservation of an entry in the current color table, depending on the value of the `reserve` parameter. A reserved entry cannot be matched by another application's search function, and `Color2Index` (or other functions that depend on it such as `RGBForeColor`, `RGBBackColor`, and `SetCPixel`) never return that entry to another client. You could use this function to selectively protect a color for color table animation.

The `ReserveEntry` function copies the low byte of the `gdID` field of the current `GDevice` data structure into the low byte of the `ColorSpec.value` field of the color table when reserving an entry, and leaves the high byte alone. `ReserveEntry` acts like selective protection and does not allow any changes if the current `gdID`

field is different than the one in the `ColorSpec.value` field of the reserved entry. If a requested match is already reserved, `ReserveEntry` returns a protection error. It can remove reservation from any entry.

SetEntries

You can use the `SetEntries` function to set a group of color table entries for the current `GDevice` data structure.

```
pascal void SetEntries (
                        short start,
                        short count,
                        cSpecArray aTable);
```

`start` The index of the first entry to be changed.

`count` The number of entries to be changed.

`aTable` An array of `ColorSpec` data structures containing the colors to be used.

DESCRIPTION

The `SetEntries` function sets a group of color table entries for the current `GDevice` data structure, starting at a given position for the specified number of entries. Use the `aTable` parameter to directly specify a `cSpecArray` structure, not the beginning of a color table. The `ColorSpec.value` fields of the entries must be in the logical range for the target device's assigned pixel depth. Thus, with a 4-bit pixel size, the `ColorSpec.value` fields should be in the range 1 to 15. With an 8-bit pixel size, the range is 0 to 255. Note that all values are zero-based; for example, to set three entries, pass 2 in the count parameter.

▲ **WARNING**
Instead of using `SetEntries`, you should use the Palette Manager function `SetEntryColor` to allow your application to run in a multiscreen or multitasking environment. ▲

The `SetEntries` positional information works in logical space rather than in the actual memory space used by the hardware. Requesting a change at the fourth position in the color table may not modify the fourth color table entry in the

hardware, but it does correctly change the color on the screen for any pixels with a value of 4 in the video card. The `SetEntries` mode characterized by a start position and a length is called ***sequence mode.*** In this case, `SetEntries` sequentially loads new colors into the hardware in the same order as they appear in the `aTable` parameter, copies the `clientID` fields for changed color table entries from the current `GDevice` data structure's `gdID` field, and ignores the `ColorSpec.value` fields.

The other `SetEntries` mode is called ***index mode.*** It allows the `cSpecArray` structure to specify where the data will be installed on an entry-by-entry basis. To use this mode, pass –1 for the start position, with a valid count and a pointer to the `cSpecArray` data structure. Each entry is installed into the color table at the position specified by the `ColorSpec.value` field of each entry in the `cSpecArray` data structure. In the current `GDevice` data structure's color table, the `ColorSpec.value` fields of all changed entries are assigned the `GDevice` data structure's `gdID` value.

When the Color Manager changes color table entries, it invalidates all cached fonts, and changes the color tables's seed number so that the next drawing operation triggers the Color Manager to rebuild the inverse table. If any of the requested entries are protected or out of range, the Color Manager returns a protection error, and nothing happens. The Color Manager changes a reserved entry only if the current `gdID` field of the current `GDevice` data structure matches the low byte of the intended `ColorSpec.value` field in the color table.

SaveEntries

You can use the `SaveEntries` function to save a selection of color table entries.

```
pascal void SaveEntries (
                    CTabHandle srcTable,
                    CTabHandle resultTable,
                    ReqListRec *selection);
```

`srcTable` The color table containing entries to be saved.

`resultTabl1` The color table in which to save the entries.

`selection` The entries to be saved, as indicated not by a range of indexes, but by a special structure noted in the description.

DESCRIPTION

The SaveEntries function saves a selection of color table entries from the srcTable parameter in the resultTable parameter. The entries to be set are enumerated in the selection parameter, which uses the ReqListRec data structure described on page 5-5. (These values are offsets into a ColorTable data structure, not the contents of the ColorSpec.value field.)

If an entry is not present in srcTable, then SaveEntries sets that position of the selection parameter to colReqErr, and that position of resultTable contains random values. If SaveEntries can't find one or more entries, then it posts an error code to QDError; however, for every entry in selection which is not colReqErr, the values in resultTable are valid. SaveEntries assumes that the color table specified by the srcTable parameter and the request list specified by the selection parameter have the same number of entries.

SaveEntries optionally allows nil as the value of its source color table parameter. If you supply nil, SaveEntries uses the current device's color table as the source. The output of SaveEntries is the same as the input for RestoreEntries, except for the order.

RestoreEntries

You can use the RestoreEntries function to set a selection of color table entries.

```
pascal void RestoreEntries (
                    CTabHandle srcTable,
                    CTabHandle dstTable,
                    ReqListRec *selection);
```

srcTable — The color table containing entries to be restored.

dstTable — The color table in which to restore the entries.

selection — The entries to be restored, as indicated not by a range of indexes, but by a special structure noted in the SaveEntries description.

DESCRIPTION

The `RestoreEntries` function sets a selection of color table entries from the `srcTable` parameter into the `dstTable` parameter, but doesn't rebuild the inverse table. You enumerate the `dstTable` entries to be set in the `selection` parameter, which uses the `ReqListRec` data structure shown on page 5-5. (These values are offsets into `srcTable`, not the contents of the `ColorSpec.value` field.)

If a request is beyond the end of the destination color table, `RestoreEntries` sets that position in the `requestList` data structure to `colReqErr`, and returns an error. `RestoreEntries` assumes that the color table specified by the `srcTable` parameter and the request list specified by the `selection` parameter have the same number of entries.

If `dstTbl` is `nil`, or points to the current `GDevice` data structure's color table, `RestoreEntries` changes the device's color table and the hardware CLUT to these new colors. `RestoreEntries` does not change the color table's seed, so no invalidation occurs (which may cause `RGBForeColor` to act strangely). `RestoreEntries` ignores protection and reservation of color table entries.

SPECIAL CONSIDERATIONS

You generally should use the Palette Manager to give your application its own set of colors; use of `RestoreEntries` should be limited to special-purpose applications. `RestoreEntries` allows you to change a color table without changing its `ctSeed` field. You can execute the application code and then use `RestoreEntries` to put the original colors back in. However, in some cases things in the background may appear in the wrong colors, since they were never redrawn. To void this, your application must build its own new inverse table and redraw the background. If you then use `RestoreEntries`, you should call the `CTabChanged` function to clean up correctly.

Operations on Search and Complement Functions

These functions enable specialized applications to add and remove custom search and complement functions to the current graphics device's list of functions, and to identify your application to the custom functions.

AddSearch

You can use the `AddSearch` function to add a function to the head of the current `GDevice` data structure's list of search functions. `AddSearch` creates and allocates an `SProcRec` data structure, which is defined page 5-4.

```
pascal void AddSearch (ColorSearchProcPtr searchProc);
```

`searchProc` A pointer to your custom search function (described on page 5-17).

AddComp

You can use the `AddComp` function to add a function to the head of the current device data structure's list of complement functions. `AddComp` creates and allocates a `CProcRec` data structure, which is described on page 5-4.

```
pascal void AddComp (ColorComplementProcPtr compProc);
```

`compProc` A pointer to your complement function, as described on page 5-18.

DelSearch

You can use the `DelSearch` function to remove a custom search function from the current `GDevice` data structure's list of search functions. `DelSearch` disposes of the chain element, but does nothing to the `ProcPtr` data structure.

```
pascal void DelSearch (ColorSearchProcPtr searchProc);
```

`searchProc` A pointer to the custom search function (described on page 5-17) to be deleted.

DelComp

You can use the `DelComp` function to remove a custom complement function from the current `GDevice` data structure's list of complement functions. `DelComp` disposes of the chain element, but does nothing to the `ProcPtr` data structure.

```
pascal void DelComp (ColorComplementProcPtr compProc);
```

`compProc` A pointer to the complement function (described on page 5-18) to be deleted.

SetClientID

You can use the `SetClientID` function to set the `gdID` field in the current `GDevice` data structure to identify this client program to its search and complement functions.

```
pascal void SetClientID (short id);
```

`id` The ID to be set in the device data structure.

Application-Defined Functions

By creating a custom search function, your application can override the Color Manager's code for inverse table mapping. By creating a custom complement function, your application can override the Color Manager's color inversion method.

MySearchProc

By creating a custom search function, your application can override the Color Manager's code for inverse table mapping. Your `MySearchProc` function should

examine the `RGBColor` data structure passed to it by the Color Manager and return the index to the best-mapping color in the current `GDevice` data structure.

```
pascal Boolean MySearchProc (
                    RGBColor *rgb,
                    long *position);
```

`rgb` The `RGBColor` data structure passed to your search function

`position` The index of the best-mapping color your function finds.

DESCRIPTION

The Color Manager specifies the desired color in the `RGBColor` field of a `ColorSpec` data structure and passes it by a pointer on the stack. Your function should return the corresponding index in the `ColorSpec.value` field. If your function can't handle the search, return `false` as the function value, and pass the `RGBColor` data structure back to the Color Manager in the `rgb` parameter.

The Color Manager calls each search function in the list until one returns the Boolean value `true`. If no search function installed in the linked list returns `true`, the Color Manager calls the default search function.

MyCompProc

By creating a custom complement function, your application can override the Color Manager's color inversion method. Your `MyCompProc` color inversion function should invert the `RGBColor` data structure passed to it in the `rgb` parameter, and return the inverted value in that parameter.

```
pascal void MyCompProc (RGBColor *rgb);
```

`rgb` The `RGBColor` data structure passed to your function.

Glossary

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.

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.

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 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 picker Code, implemented as a component, that allows users to select a color from a range of possible colors.

This document was created with FrameMaker 4.0.4

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 non-device profiles.

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.

explict color A color that specifies an index value in the devices 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.

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.*

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.*

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.

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.

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.

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 specificies 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.

RGB space A three-dimensional color space whose components are the red, green, and blue intensities that make up a given color.

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.**

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.

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.

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.

Glossary

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.

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.

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 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 picker Code, implemented as a component, that allows users to select a color from a range of possible colors.

This document was created with FrameMaker 4.0.4

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 non-device profiles.

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.

explict color A color that specifies an index value in the devices 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.

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.*

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.*

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.

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.

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.

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 specificies 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.

RGB space A three-dimensional color space whose components are the red, green, and blue intensities that make up a given color.

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.**

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.

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.

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.

Index

A

ActivatePalette function 1-13
AddComp function 5-16
AddPickerToDialog function 2-40
AddSearch function 5-16
AnimateEntry function 1-21
AnimatePalette function 1-22
animating palettes 1-20 to 1-22
Apple CMM enumeration 3-9
Apple profile header data structure 3-26
application-defined functions
  MyColorChangedFunction 2-60
  MyCompProc 5-18
  MyPickerFilerFunction 2-59
  MySearchProc 5-18
application-owned dialog box structure 2-26

C

callback function 3-134
CMCloseProfile function 3-51
CMCopyProfile function 3-55
CMCountProfileElements function 3-61
CMDisposeProfileSearch function 3-104
CMEnableMatchingComment function 3-77
CMFixedXYZToXYZ function 3-117
CMFlattenProfile function 3-58
CMGetColorSyncFolderSpec function 3-130
CMGetCWInfo function 3-87
CMGetIndProfileElement function 3-67
CMGetIndProfileElementInfo function 3-66
CMGetPartialProfileElement function 3-65
CMGetProfileElement function 3-62
CMGetProfileHeader function 3-64
CMGetPS2ColorRendering function 3-128
CMGetPS2ColorRenderingIntent function 3-126
CMGetPS2ColorRenderingVMSize function 3-129
CMGetPS2ColorSpace function 3-125
CMGetScriptProfileDescription function 3-74
CMGetSystemProfile function 3-100
CMHLSToRGB function 3-120
CMHSVToRGB function 3-122
CMLabToXYZ function 3-109
CMLuvToXYZ function 3-112
CMM check bitmap colors function 4-19
CMM check colors function 4-12
CMM check pixel map colors function 4-27
CMM component interface version constant 4-3
CMM concatenated profiles initialization function 4-22
CMM create device-linked profile function 4-29
CMM information data structure 3-32
CMM initialization function 4-9
CMM match bitmap colors function 4-16
CMM match colors function 4-11
CMM match pixel map colors function 4-24
CMM PostScript color rendering function 4-35
CMM PostScript color rendering intent function 4-33
CMM PostScript color space function 4-31
CMM PostScript CRD VM size function 4-38
CMM profile flattening function 4-40
CMM profile unflattening function 4-42
CMM profile validation function 4-15
CMMs, obtaining information about 3-87
CMNewProfile function 3-53
CMNewProfileSearch function 3-101
CMOpenProfile function 3-50
CMProfileElementExists function 3-61
CMRemoveProfileElement function 3-73
CMRGBToGray function 3-124
CMRGBToHLS function 3-118
CMRGBToHSV function 3-121
CMSearchGetIndProfileFileSpec function 3-105

This document was created with FrameMaker 4.0.4

CMSearchGetIndProfile function 3-104
CMSetPartialProfileElement function 3-70
CMSetProfileElement function 3-71
CMSetProfileElementReference function 3-73
CMSetProfileElementSize function 3-69
CMSetProfileHeader function 3-72
CMSetSystemProfile function 3-99
CMUnflattenProfile function 3-59
CMUpdateProfile function 3-52
CMUpdateProfileSearch function 3-102
CMValidateProfile function 3-57
CMXYZToFixedXYZ function 3-116
CMXYZToLab function 3-108
CMXYZToLuv function 3-111
CMXYZToYxy function 3-113
CMY2RGB function 2-54
CMY color data structure 3-39
CMY color structure 2-35
CMYK color data structure 3-38
CMYxyToXYZ function 3-114
Color2Index function 5-5
color-changed functions 2-19, 2-60
color checking
  checking a bitmap 3-95
  checking a pixel map 3-90
color conversion
  from fixed XYZ to XYZ 3-117
  from HLS to RGB 3-120
  from HSV to RGB 3-122
  from L*a*b* to XYZ 3-109
  from L*u*v* to XYZ 3-112
  from RGB to Gray 3-124
  from RGB to HLS 3-118
  from RGB to HSV 3-121
  from XYZ to fixed XYZ 3-116
  from XYZ to L*a*b* 3-108
  from XYZ to L*u*v* 3-111
  from XYZ to Yxy 3-113
  from Yxy to XYZ 3-114
color-conversion-component function selectors enumeration 3-20
color conversion component version constant 3-22
ColorInfo data type 1-6
Color Manager 5-3 to 5-18
  application-defined functions for 5-17 to 5-18
  constants and data types in 5-3 to 5-5
  functions in 5-5 to 5-17
color matching
  concluding a high-level session 3-77
  creating a color world for 3-81
  creating a concatenated color world for 3-82
  disposing of a color world 3-86
  matching a bitmap 3-92
  matching a list of colors 3-97
  matching a pixel map 3-88
  setting up a high-level session 3-75
  turning on or off 3-77
  using embedded profiles 3-78
  using low-level functions 3-80
color models, conversion between 2-54 to 2-57
color packing enumeration 3-14
color picker–defined functions
  MyColorPickerDispatch 2-61
  MyDoEdit 2-79
  MyDoEvent 2-77
  MyDrawPicker 2-76
  MyExtractHelpItem 2-75
  MyGetColor 2-66
  MyGetDialog 2-64
  MyGetEditMenuState 2-74
  MyGetIconData 2-69
  MyGetItemList 2-64
  MyGetProfile 2-72
  MyGetPrompt 2-70
  MyInitPicker 2-63
  MyItemHit 2-78
  MySetBaseItem 2-68
  MySetColor 2-67
  MySetOrigin 2-71
  MySetProfile 2-73
  MySetPrompt 2-70
  MySetVisibility 2-65
  MyTestGraphicsWorld 2-62
Color Picker Manager 2-5 to 2-80
  application-defined functions for 2-58 to 2-60
  color picker–defined functions for 2-60 to 2-80
  constants and data structures in 2-5 to 2-36
  functions in 2-36 to 2-58
  result codes in 2-80

color picker parameter block 2-20
color pickers
  color-changed functions for 2-60
  event filter functions for 2-18 to 2-19, 2-59
colors
  in a palette 1-6
color spaces enumeration 3-15
color space signatures enumeration 3-13
ColorSync 1.0 element tag signatures enumeration 3-22
ColorSync data-transfer function command enumeration 3-9
ColorSync Manager bitmap data structure 3-42
ColorSync Manager gestalt selectors enumeration 3-7
color types enumeration 2-7
color union data structure 3-40
color world information data structure 3-31
color world reference data structure 3-44
concatenated profile set data structure 3-30
CopyPalette function 1-23
courteous colors, PmBackColor and 1-17
CRD virtual memory size tag data structure 3-48
CreateColorDialog function 2-38
CreatePickerDialog function 2-39
CTabToPalette function 1-25
CWCheckBitMap function 3-95
CWCheckColors function 3-98
CWCheckPixMap function 3-90
CWConcatColorWorld function 3-82
CWDisposeColorWorld function 3-86
CWMatchBitmap function 3-92
CWMatchColors function 3-97
CWMatchPixMap function 3-88
CWNewLinkProfile function 3-84

D

data-transfer function 3-131
DelComp function 5-17
DelSearch function 5-16
dialog placement specifiers enumeration 2-8
DisposeColorPicker function 2-44
DisposePalette function 1-10
DoPickerDraw function 2-47
DoPickerEdit function 2-46
DoPickerEvent function 2-45

E

editing data structure 2-29
edit menu items structure 2-19
edit menu operations enumeration 2-7
edit menu state structure 2-20
EntryToIndex function 1-30
entry usage, setting 1-29
event data structure 2-27
event filter function 2-18
event filter functions (for color pickers) 2-18 to 2-19, 2-59
event forecasters enumeration 2-11
ExtractPickerHelpItem function 2-51

F

file specification location data structure 3-24
Fix2SmallFract function 2-57
fixed XYZ color data structure 3-35
format conventions xiv to xv

G

gestaltColorMatchingVersion selector 3-7
GetColor function 2-37
GetCTSeed function 5-10
GetEntryColor function 1-27
GetEntryUsage function 1-27
GetNewPalette function 1-8
GetPalette function 1-14
GetPaletteUpdates function 1-15
GetPickerColor function 2-49
GetPickerEditMenuState function 2-50
GetPickerOrigin function 2-43

GetPickerProfile function 2-53
GetPickerVisibility function 2-42
GetSubTable function 5-8
Gray color data structure 3-39

H

handle specification data structure 3-25
help item structure 2-33
HiFi color data structure 3-39
high-level color-matching-session reference data structure 3-44
HLS color data structure 3-37
HSL2RGB function 2-55
HSL color structure 2-34
HSV2RGB function 2-56
HSV color data structure 3-38
HSV color structure 2-34

I

Index2Color function 5-6
index mode 5-13
InitPalettes function 1-7
InitPalettes procedure 1-7
InvertColor function 5-7
item hit modifiers enumeration 2-7
item hit structure 2-31

L

L*a*b* color data structure 3-36
LS 3-37
L*u*v* color data structure 3-36

M

Macintosh Programmer's Workshop xv
MakeITable function 5-9
MyCMBitmapCallBackProc function 3-134
MyCMCheckColors function 4-13
MyCMCheckPixMap function 4-27
MyCMMatchBitmap function 4-16
MyCMMatchColors function 4-11
MyCMMatchPixMap function 4-24
MyCMMCheckBitmap function 4-19
MyCMMFlattenProfile function 4-40
MyCMMGetPS2ColorRendering function 4-36
MyCMMGetPS2ColorRenderingVMSize function 4-38
MyCMMGetPS2ColorSpace function 4-31
MyCMMUnflattenProfile function 4-42
MyCMMValidateProfile function 4-15
MyCMNewLinkProfile function 4-30
MyCMProfileFilterProc function 3-136
MyColorChangedFunction function 2-60
MyColorPickerDispatch function 2-61
MyColorSyncDataTransfer function 3-131, 3-132
MyCompProc function 5-18
MyDoEdit function 2-79
MyDoEvent function 2-77
MyDrawPicker function 2-76
MyExtractHelpItem function 2-75
MyGetColor function 2-66
MyGetDialog function 2-64
MyGetEditMenuState function 2-74
MyGetIconData function 2-69
MyGetItemList function 2-64
MyGetProfile function 2-72
MyGetPrompt function 2-70
MyInitNCMM function 4-9
MyInitPicker function 2-63
MyItemHit function 2-78
MyPickerFilterFunction function 2-59
MySearchProc function 5-18
MySetBaseItem function 2-68
MySetColor function 2-67
MySetOrigin function 2-71
MySetProfile function 2-73
MySetPrompt function 2-70
MySetVisibility function 2-65
MyTestGraphicsWorld function 2-62

N

NCMBeginMatching function 3-75, 3-77
NCMDrawMatchedPicture function 3-78
NCMUseProfileComment function 3-79
NCWNewColorWorld function 3-81
NewPalette function 1-9
NSetPalette function 1-12

P

Palette data type 1-5
Palette Manager
  See also palettes 1-6
  allocation of colors 1-6
  functions 1-6 to 1-30
  initialization 1-7
palette resource 1-30
palettes
  animating 1-20 to 1-22
  animating entries 1-21
  colors of 1-6
  creating 1-8 to 1-11
  drawing with 1-16 to 1-20
  manipulating 1-23 to 1-26
  manipulating entries 1-27 to 1-30
  modifying 1-27 to 1-30
  record format 1-5
  resource format 1-30
  resources 1-30
  setting entry usage 1-29
  usage categories 1-3
  and windows 1-11 to 1-16
PaletteToCTab function 1-26
PickColor function 2-36
picker actions enumeration 2-5
picker attributes 2-10
picker color structure 2-16
picker flags 2-8
picker icon structure 2-17
picker initialization structure 2-18
picker message request codes 2-12
picker messages enumeration 2-12
picker-owned dialog box structure 2-25
picker structure 2-17
picture comment IDs enumeration 3-10
picture comment selectors enumeration 3-11
'pltt' resource 1-30
PmBackColor function 1-17
PmForeColor function 1-16
PMgrVersion function 1-7
pointer specification data structure 3-25
PostScript
  obtaining profile data for 3-125 to 3-130
PrGeneral function operation codes enumeration 3-22
profile 2.0 header data structure 3-26
profile classes enumeration 3-8
profile header for ColorSync 1.0 data structure 3-45
profile location data structure 3-24
profile location type enumeration 3-5
profile location union data structure 3-23
profile reference abstract data structure 3-43
profiles
  adding a reference tag 3-73
  checking elements of 3-61
  closing 3-51
  copying 3-55
  counting elements of 3-61
  creating 3-53
  creating a device-linked profile 3-84
  embedding in a picture 3-79
  flattening 3-58
  folder, locating 3-130
  getting an element of 3-62
  getting a partial element of 3-65
  getting location of 3-56
  getting the header of 3-64
  getting the system profile 3-100
  obtaining an element's tag and size 3-66
  obtaining element data 3-67
  obtaining the name and script code of 3-74
  opening 3-50
  removing an element of 3-73
  reserving an element's data size 3-69
  searching the contents of 3-101 to 3-106
  setting or replacing element data 3-71

setting partial element data 3-70
setting the header 3-72
setting the system profile 3-99
unflattening 3-59
updating 3-52
validating 3-57
profile search record data structure 3-33
profile search result reference abstract data structure 3-44
`ProtectEntry` function 5-10

R

`RealColor` function 5-7
rendering intent values enumeration 3-19
request codes
optional, constants for 4-5
required, constants for 4-4
request codes for color pickers 2-12
`ReserveEntry` function 5-11
`ResizePalette` function 1-24
resources
palette 1-30
`'pltt'` 1-30
`RestoreBack` function 1-20
`RestoreDeviceClut` function 1-24
`RestoreEntries` function 5-14
`RestoreFore` function 1-18
`RGB2CMY` function 2-55
`RGB2HSL` function 2-56
`RGB2HSV` function 2-57
RGB color data structure 3-37

S

`SaveBack` function 1-19
`SaveEntries` function 5-13
`SaveFore` function 1-18
sequence mode 5-13
`SetClientID` function 5-17
`SetEntries` function 5-12
`SetEntryColor` function 1-28
`SetEntryUsage` function 1-29
`SetPalette` function 1-11
`SetPaletteUpdates` function 1-15
`SetPickerColor` function 2-48
`SetPickerOrigin` function 2-44
`SetPickerProfile` function 2-52
`SetPickerPrompt` function 2-42
`SetPickerVisibility` function 2-41
`SmallFract2Fix` function 2-58
system-owned dialog box structure 2-24

U

usage categories in palettes 1-3

W

Window Manager, interactions with Palette Manager 1-11 to 1-16

X

XYZ color component data structure 3-35
XYZ color data structure 3-35

Y

Yxy color data structure 3-37

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Line art was created using Adobe Illustrator™ and Adobe Photoshop™.

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.

WRITERS
Judy Melanson, Tony Francis, Michael Kline, Rob Dearborn

DEVELOPMENTAL EDITORS
Jeanne Woodward, Beverly McGuire

ILLUSTRATORS
Bruce Lee, Ruth Anderson, Lisa Hymel

PRODUCTION EDITORS
Lorraine Findlay, Alex Solinski

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.

Acknowledgment to Richard Collyer. Fdgar Lee, David Van Brink, Wei-Ling Chu, Han Nguyen, Forrest Tanaka, John Myer, Josh Weisberg, John Wang, Shannon Holland, and Dave Johnson.

This document was created with FrameMaker 4.0.4